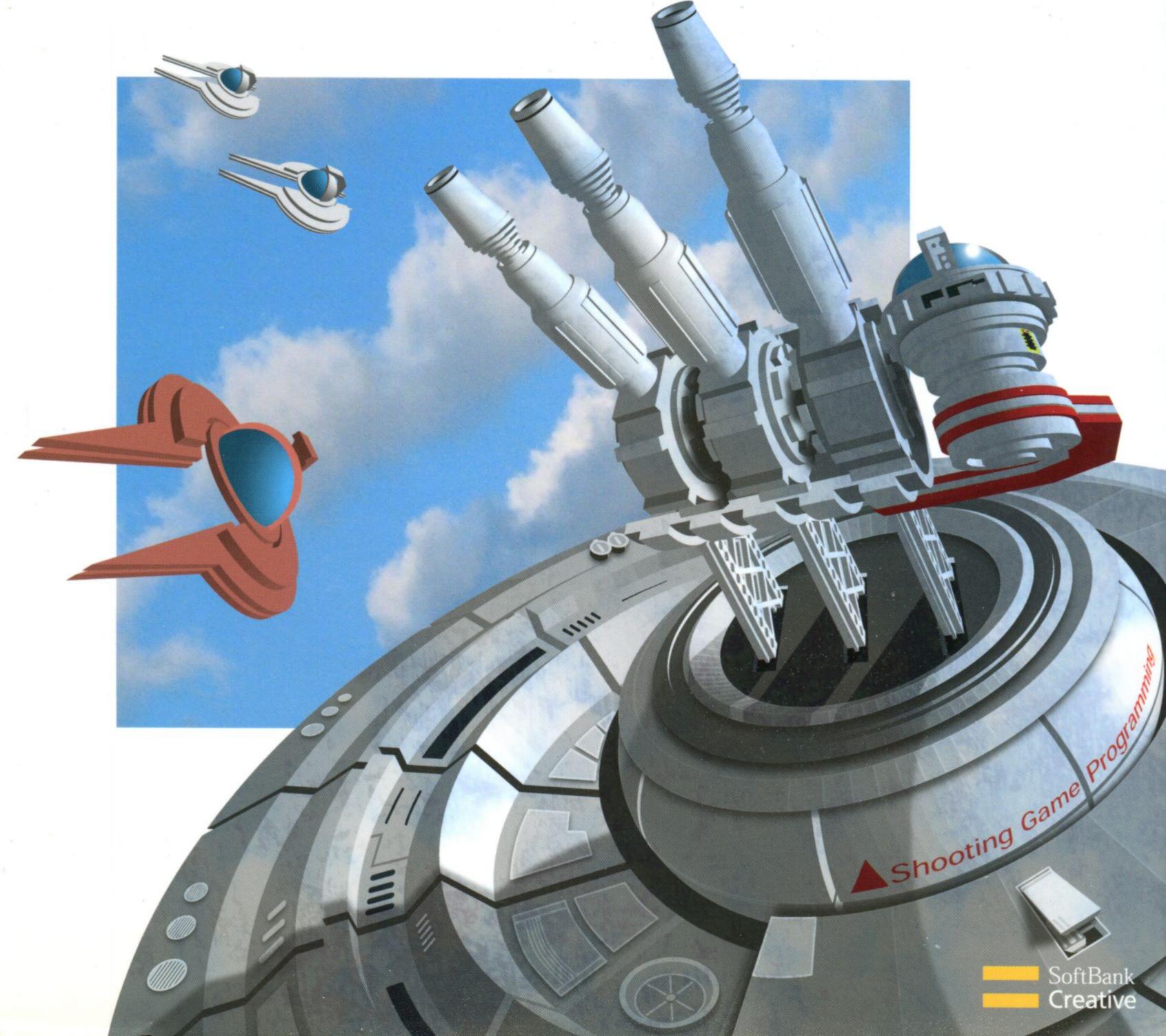
# シューティングゲーム
Shooting Game Programming
# プログラミング

松浦健一郎／司ゆき 著

SoftBank Creative

# 付録CD-ROM
## サンプルシューティングゲーム「紫雨（むらさめ）」

ショット、ビーム、4種類のボムと2つの特殊攻撃を駆使して敵を倒せ！
難易度調整、リプレイ、ハイスコア表示機能を搭載

# シューティングゲームプログラミング

Shooting Game Programming

松浦健一郎／司ゆき 著

SoftBank Creative

本書は、月刊『C MAGAZINE』に掲載された連載記事「ゲーム・ノ・シクミ シューティングゲーム編」(2005年3月号～2006年1月号)の内容を加筆・修正し、それをベースとして再構成したものです。

本書に関する情報をインターネットでも公開しています。
以下のURLにアクセスしてください。
http://www.sbcr.jp/media/book/

# はじめに

本書はシューティングゲームの制作方法を実践的かつ平易に解説する本です。これからシューティングゲームを作ろうという方にはもちろん、シューティングゲームを題材にプログラミングを学ぼうという方にもお使いいただけます。本書は次のような目的にお勧めです。

・シューティングゲームを自分で作りたい
・シューティングゲームがどのような手法を使って作られているのかを知って、ゲームをもっと楽しみたい
・C++やDirectXによるプログラミングを学びたい
・自作のシューティングゲームをもっと魅力的にしたい
・学校の課題でシューティングゲームを作ることになったが、どこから手をつけたらよいのかわからない
・ビームやボム、かすりやソードといった、シューティングゲームに使えるさまざまな要素の実現方法が知りたい
・せっかく新しいPCを買ったので何かソフトウェアを開発したい
・サンプルゲームを遊びながら、シューティングゲームの構想をじっくりと練りたい
・『シューティングゲームアルゴリズムマニアックス』を読んで、シューティングゲームの骨格部分のプログラミングにもヒントがほしくなった
・とにかくシューティングゲームが好きだ

本書はごく単純なプログラムから始めて、自機・弾・敵・ステージ・ボス・アイテム・特殊攻撃といったさまざまな要素を追加し、本格的なシューティングゲームへと育てていきます。解説は極力平易にしたので、リラックスした気分でお読みください。一方で、本格的なシューティングゲームのプログラミングに欠かせないDirectXの知識や、多数のキャラクターを効率的に管理するためのタスクシステムについても、詳細に解説しています。

誌面だけではなく、付録CD-ROMもぜひお楽しみください。各章の内容に対応した実際に遊べるサンプルゲームのほか、ついつい筆が進みすぎて誌面から飛び出した「リプレイ機能」「難易度選択機能」「データファイルのアーカイブ化」といったプログラミング情報も満載です。

プログラマーの方にはもちろん、プレイヤーの方にとってもゲームデザイナーの方にとっても、本書がシューティングゲームの世界をより深く楽しむためのガイドブックになれば幸いです。

2006年秋　松浦　健一郎/司 ゆき

Shooting Game Programming

# CONTENTS

## Chapter 1 シューティングゲームを作るには



## Chapter 2 ゲームライブラリ

## Chapter 3 タスクシステム

## Chapter 4 自機

## Chapter 5 弾

## Chapter 6 敵



## Chapter 7 ゲームの外枠

## Chapter 8 ステージ

## Chapter 9 ボス



## Chapter 10 アイテムとパワーアップ

## Chapter 11　特殊攻撃



## Chapter 12　オリジナルゲーム制作ガイド

## Appendix 付録



※「ゲームのブラッシュアップ」では、リプレイ機能、難易度の選択、データファイルのアーカイブ化について紹介しています。なお、これらに関しては、付録CD-ROMに収録した「ChapterA.pdf」にて詳しく解説しています。

# Chapter 01

# シューティングゲームを作るには

「シューティングゲームが作りたい！」そう思い立っても、どこから手をつければよいのか、最初はなかなかわからないものです。派手な攻撃や緻密な弾幕のアイディアが浮かんでいたら、なおさら早くそれを実際に動く形にしたいでしょう。
本章では、「シューティングゲームとは何か」「シューティングゲームを作るにはどんな作業が必要なのか」「どんな開発環境を使えばよいのか」といった点を整理して、いち早くゲームの制作を始めるための準備を整えます。

shooting game programming

# シューティングゲームとは何か

ご存じのとおりシューティングゲームとは、敵や弾をかいくぐり、ショットやレーザーを撃ってスコアを稼ぐゲームです。弾幕や演出やスコア稼ぎの方法など、いろいろな工夫はありますが、「避けて、撃つ」という基本ルールはほとんどのゲームに共通です。

ハードウェアの進化に伴ってグラフィックやサウンドは派手になりましたが、ゲームの内容は今でも比較的シンプルです。ほとんどのゲームは、「8方向スティック＋ショットボタン＋特殊攻撃（ボム）ボタン」という昔ながらの操作系を採用しているため、初めて遊ぶときでもすぐに楽しむことができます。

世の中には膨大な数のシューティングゲームがあり、各ゲームにはそれぞれ独自の仕掛けが詰め込まれています。遊び込むほどに、作品ごとに異なったゲーム性や攻略法が見えてくることも、シューティングゲームの大きな魅力です。

## 本書で解説すること

ひとくちにシューティングゲームといっても、実に多くの種類があります。しかし、基本は同じで、その基本の上に築かれたさまざまな仕掛けの部分が、それぞれの作品の個性となっています。本書では主に、すべてのシューティングゲームに共通する基本の部分を作るための方法を解説します。基本の部分が作れるようになれば、仕掛けの部分を作ることは技術的には難しくありません。むしろ重要なのはアイディアなので、この部分はぜひ皆さんにおまかせしたいと思います。

なお、本書ではいくつかのアイディアの実現方法も解説しています。これらは、拙著『シューティングゲームアルゴリズムマニアックス』に掲載しきれなかった仕掛けのうち、読者の皆さまからご要望をいただいたものです。より多彩な仕掛けのアルゴリズムやソースコードに関しては、『シューティングゲームアルゴリズムマニアックス』もご利用いただければ幸いです。同書では、古今東西のアーケードゲームを数多く題材に取り上げ、特徴のある仕掛けについて図とソースコードでわかりやすく解説を加えています。ぜひご利用ください。『シューティングゲームアルゴリズムマニアックス』については、以下のWebサイトをご参照ください。

**シューティングゲームアルゴリズムマニアックス**

http://isbn.sbcr.jp/27316

http://www.cmagazine.jp/books/stg/index.html

### プログラマーとシューティングゲーム

最近のゲームは画像やムービーを多用していて派手だけれども、プログラマーが活躍する場が少なくなって寂しい……と感じることがあります。特にRPGやアドベンチャーゲームでは、プログラムよりも絵や音や動画といったコンテンツの方が注目されがちです。

その一方で、シューティングゲームは、まだまだプログラマーが活躍する場面の多いジャンルだといえます。魅力的な動きの敵や弾を作ったり、ゲームバランスを細かく調整したりするためには、プログラマーの力が不可欠です。

画像やムービーを多用しなくても工夫しだいで面白いゲームになるということも、シューティングゲームの特色です。そのため、ホビープログラマーが1人または少人数でゲームを作る場合、シューティングゲームは都合のよいジャンルだといえます。

また、シューティングゲームはプログラミングの学習にも適しています。構造化プログラミングやオブジェクト指向プログラミングを学ぶにも、リアルタイムのグラフィック処理を学ぶにも、シューティングゲームは格好の題材です。

シューティングゲームを作ることができるようになれば、同じ技術の応用で他のさまざまなジャンルのゲームも作れるようになるでしょう。

## シューティングゲームの構成要素

シューティングゲームというと、自機を操作して弾を避け、ショットを撃って敵を破壊する……というイメージが頭に浮かびます。まずはこのイメージをもとに、シューティングゲームにはどんな要素が必要なのかを、あらためて整理してみます（Fig. 1-1）。

### 自機

プレイヤーが操作するキャラクターです。飛行機や宇宙船のことが多いですが、人間や動物のこともあります。最近は、美少女キャラクターを中心とする人型の自機が好んで使われていますが、一方で戦闘機やヘリコプターといった古典的なものも健在です。

### 敵

自機に攻撃を仕掛けてくるキャラクターです。破壊できない敵もいますが、ほとんどの敵は自機の武器を使って破壊することができます。ステージの最後でボスと呼ばれる巨大な敵を出現させるゲームもあります。最近のゲームではボスに加えて、ステージの中盤に中ボスと呼ばれる、ボスに次いで大きな敵が出現することも多くなりました。

## 武器

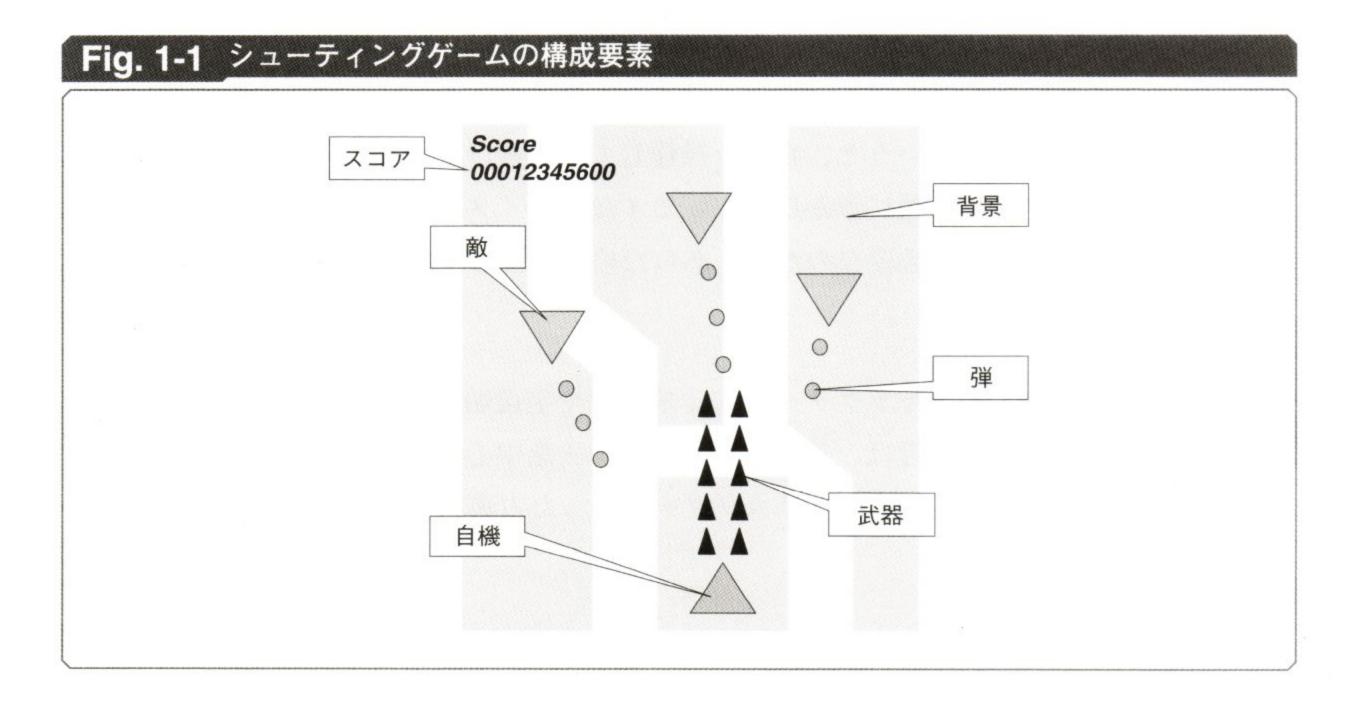
自機が敵に向かって放つ攻撃です。ショットやレーザーなどが中心ですが、武器の種類はゲームによって千差万別です。武器を敵に当てると破壊することができます。また、ショットやレーザーといった通常の武器以外に、ボムのような特殊な武器もあります。ボムの多くは、敵の弾を消したり、画面内のすべての敵にダメージを与えたりといった効果を持ちます。

## 弾

敵が自機に向かって放つ攻撃です。多くのゲームでは、武器で弾を破壊することはできません。弾に自機が接触すると破壊されてしまいます。ゲームによっては、弾にさまざまな色と形があり、それぞれ速度や大きさといった性質が異なります。また、レーザーやミサイルといった特殊な弾が出現するゲームもあります。

## 背景

自機、敵、弾の背後に表示される地面や宇宙空間などです。多くのゲームでは、背景がスクロール（ある方向に流れること）します。上下にスクロールするゲームを縦スクロールゲーム、左右にスクロールするゲームを横スクロールゲームと呼びます。縦スクロールは上から下、横スクロールは右から左にスクロールするのが一般的です。

**Fig. 1-1** シューティングゲームの構成要素

### スコア

得点です。敵を破壊したり、アイテムを拾ったりするとスコアが入ります。スコアはプレイヤーの技量を反映するので、ほとんどのシューティングゲームでは、高いスコアを稼ぐことがゲームの目的の1つになっています。敵を連続して倒したり、アイテムをもらさず回収したりといった特別な手順によって、より高いスコアを稼げるようにしているゲームもあります。

※

シューティングゲームには、こういった目に見える要素の他に、目に見えないながらも非常に重要な「当たり判定処理」という要素もあります。当たり判定処理などの目に見えない部分については後述します。

なお、本書ではシューティングゲームの構成要素について、以上のような呼び方に基づいて解説を行っていきます。

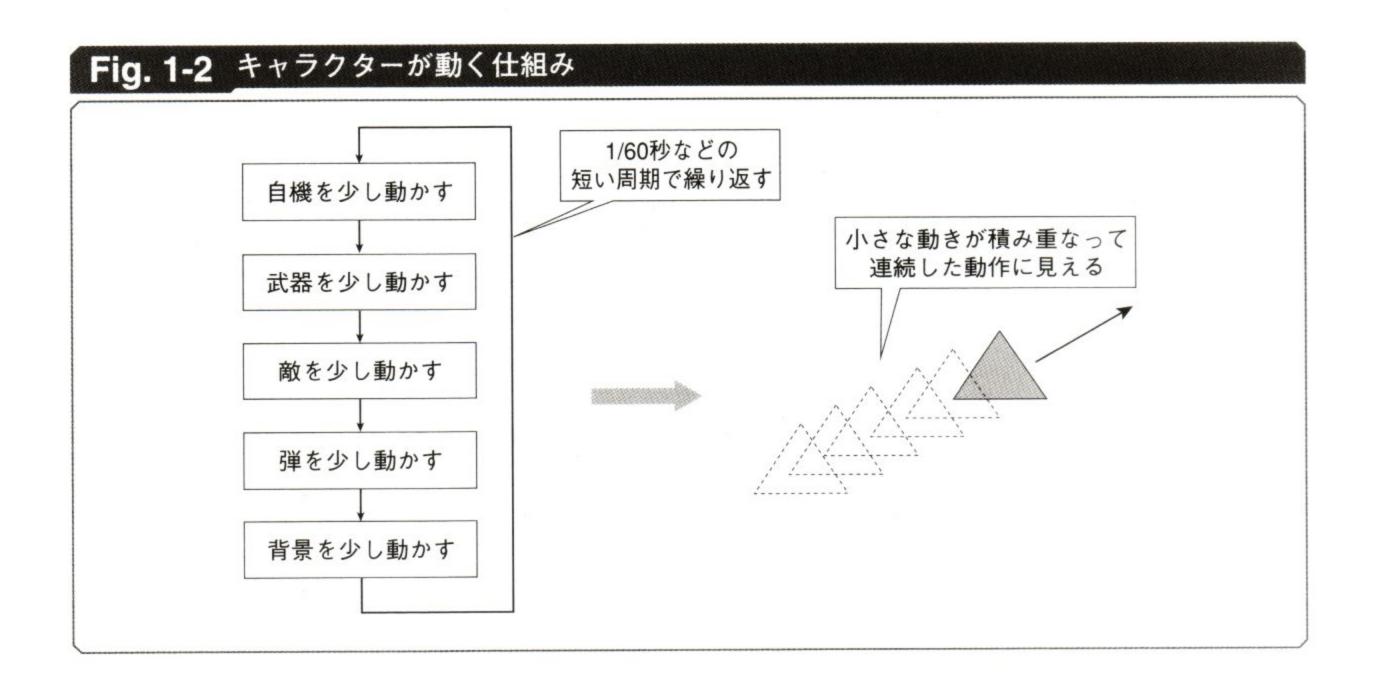
## キャラクターはどうして動くのか

シューティングゲームにかぎらず、キャラクターを動かす処理はゲームプログラムの基本です。自機や弾といったキャラクターはなめらかに動きますが、実はこの動きは小さな動きの積み重ねからできています（Fig. 1-2）。

例えば自機を右上に動かす場合には、最初に少しだけ右上に動かし、短い時間だけ待って、また少しだけ右上に動かします。こういった小さな動きを非常に短い周期で何回も繰り返すことによって、連続した動きを作り出します。

ほとんどのゲームでは、この繰り返しの周期を画面表示機器の更新周期に合わせています。これらの周期を合わせると、アニメーションをなめらかに表示することができるからです。例えば、家庭用テレビやPC用ディスプレイの一般的な画面更新周期は1/60秒なので、多くのゲームはこの周期に合わせて、1/60秒周期で進行します。

武器・敵・弾・背景についても自機と同じように、小さな動きを繰り返すことによって連続した動きを作り出します。シューティングゲームのプログラムは、自機・武器・敵・弾・背景などをそれぞれ小さく動かす処理を1周期として、これを何周期も繰り返すことで成り立っています。

**Fig. 1-2** キャラクターが動く仕組み

## フレームとは

画面全体のことを指して、フレームと呼ぶことがあります。これはテレビのような表示機器の用語ですが、ゲームでもよく使われる言葉です。

画面全体を1秒間に更新する回数のことを、「秒間30フレーム」とか「秒間60フレーム」などといいます。1/60秒周期で画面全体を更新するゲームの場合は、秒間60フレームというわけです。

シューティングゲームをはじめとするアーケードゲームの多くは、秒間60フレームです。これは、テレビやディスプレイといった画面表示機器の一般的な画面更新周期に合わせてあるのです。

キャラクターを動かすには、1フレームごとに少しずつキャラクターの座標や角度を変化させます。秒間60フレームの場合には、キャラクターを60回動かすことによって、1秒間の動きを作るというわけです。

ゲームを遊ぶときにも作るときにも、フレームという言葉は非常によく使います。例えば次のような使い方です。

「新作の3Dゲームを買ったんだけど、30フレームでがっかりだよ」

「シューティングゲームはやっぱり60フレームじゃないとなぁ！」

「なんとか頑張って1フレームに処理を押し込められない？」

なお、1秒間に更新できるフレーム数のことをフレームレートと呼びます。フレームレ

ートはfps (frames per second) という単位で表記します。秒間60フレームは、60fpsということです。ゲーマー同士の会話や、ゲームプログラマー同士の会話では、フレーム、フレームレート、fpsといった言葉を明確に区別することなく、「60フレーム」「秒間60フレーム」「フレームレートが60」「60fps」のようにいろいろな表現を使うことがあります。

また、イントという言葉もあり、フレームと同じ意味で使われることがあります。この言葉はinterrupt (インタラプト、割り込む、中断する) に由来しており、この場合は垂直同期割り込みを指しています。垂直同期割り込みは、画面更新のタイミングを知らせるためのもので、一般的なゲーム用のハードウェアでは1/60秒単位で発生します。

# 当たり判定処理とは

キャラクター同士が接触したかどうかを判定する処理のことを、当たり判定処理と呼びます。シューティングゲームでは、自機と敵、あるいは自機と弾が接触すると、自機はダメージを受けます。敵と武器が接触したときには、敵を破壊することができます。当たり判定処理によってキャラクター同士が接触したかどうかを判定し、その結果に応じて自機や敵を破壊するというわけです。

シューティングゲームにかぎらず、あらゆるジャンルのゲームにおいて当たり判定処理は非常に重要な要素です。格闘ゲームでパンチが当たったかどうかを判断するのも、スポーツゲームでボールを打ったかどうかを判断するのも、すべて当たり判定処理です。

先ほどキャラクターが動く仕組みについて説明しましたが、ただキャラクターが動くだけではゲームにはなりません。キャラクターを動かす処理に対して、当たり判定処理や破壊の処理、スコアを加算する処理などを加えることによって、初めてゲームになるのです。

## 当たり判定処理の例

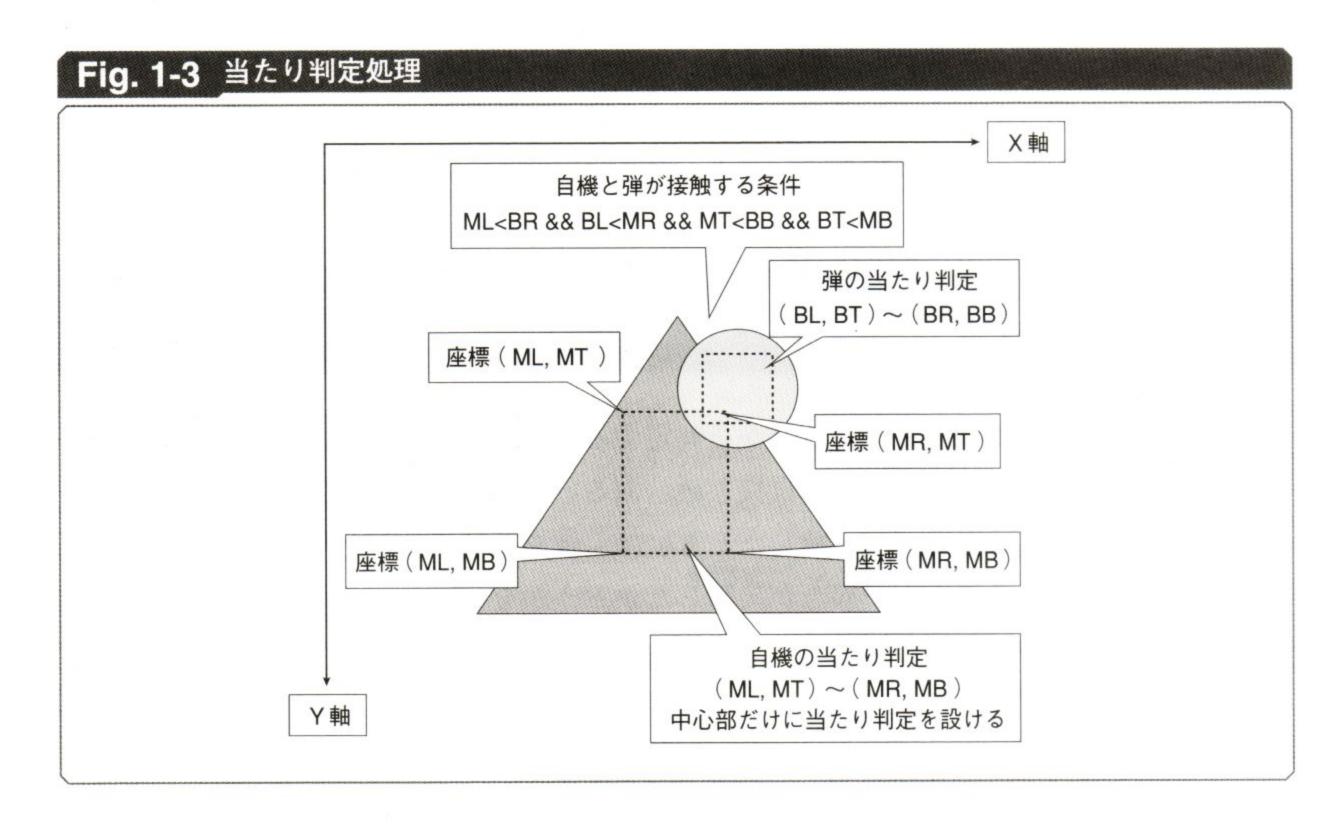
当たり判定処理にはいろいろな方法があります。オーソドックスなのは矩形を使った判定方法です (Fig. 1-3)。ゲームによっては矩形ではなく、円などを使って当たり判定処理を行う場合もあります。

なお、本書では主に2Dの当たり判定処理を扱いますが、3Dの当たり判定処理も考え方は似ています。3Dでは直方体や球を使った当たり判定処理や、ポリゴン同士の当たり判定処理を行います。

さて、本書では当たり判定処理を行うための座標情報を「当たり判定」と呼び、この情

報を使用して実際に判定を行う処理を「当たり判定処理」と呼ぶことにします。Fig. 1-3では、自機の当たり判定を（ML, MT）～（MR, MB）、弾の当たり判定を（BL, BT）～（BR, BB）としました。このとき、自機と弾とが接触する条件は次のとおりです。

```
ML<BR && BL<MR && MT<BB && BT<MB
```

**Fig. 1-3 当たり判定処理**

## 複雑な形に対する当たり判定処理

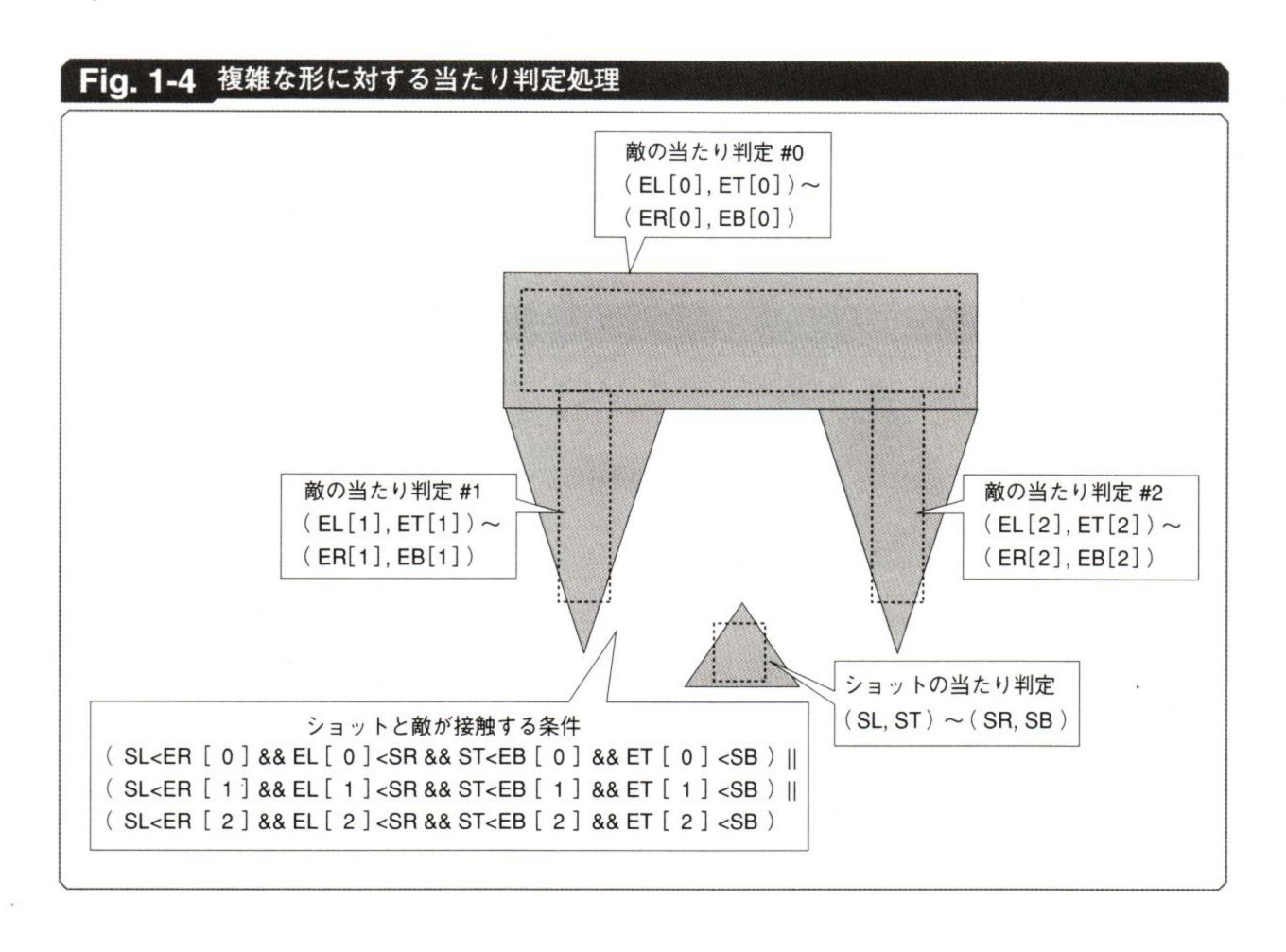
多くの場合、キャラクターの当たり判定は1個の矩形で大丈夫です。しかし、複雑な形をしたキャラクターの場合は、1個の矩形ではうまく当たり判定を表現できないことがあります。その場合は、複数の矩形を組み合わせて当たり判定を構成します（Fig. 1-4）。

当たり判定処理を行うには、次のように各矩形に対する条件式を論理和（||）で結びます。つまり、矩形のいずれかの組み合わせが重なり合っていたら、接触したと見なすわけです。

```
(SL<ER[0] && EL[0]<SR && ST<EB[0] && ET[0]<SB) ||
(SL<ER[1] && EL[1]<SR && ST<EB[1] && ET[1]<SB) ||
(SL<ER[2] && EL[2]<SR && ST<EB[2] && ET[2]<SB)
```

なお、ここでは説明のために||演算子を使いましたが、実際のプログラムではforループなどを使います。矩形の数だけforでループして、いずれかの組み合わせに関して条件が成立したら、接触したと見なすような処理にします。例えば以下のようなプログラムです。

```
int i;
for (i=0; i<3; i++) {
    if (SL<ER[i] && EL[i]<SR && ST<EB[i] && ET[i]<SB) break;
}
if (i<3) {
    // ※接触したときの処理※
}
```

**Fig. 1-4 複雑な形に対する当たり判定処理**

# 当たり判定処理の高速化

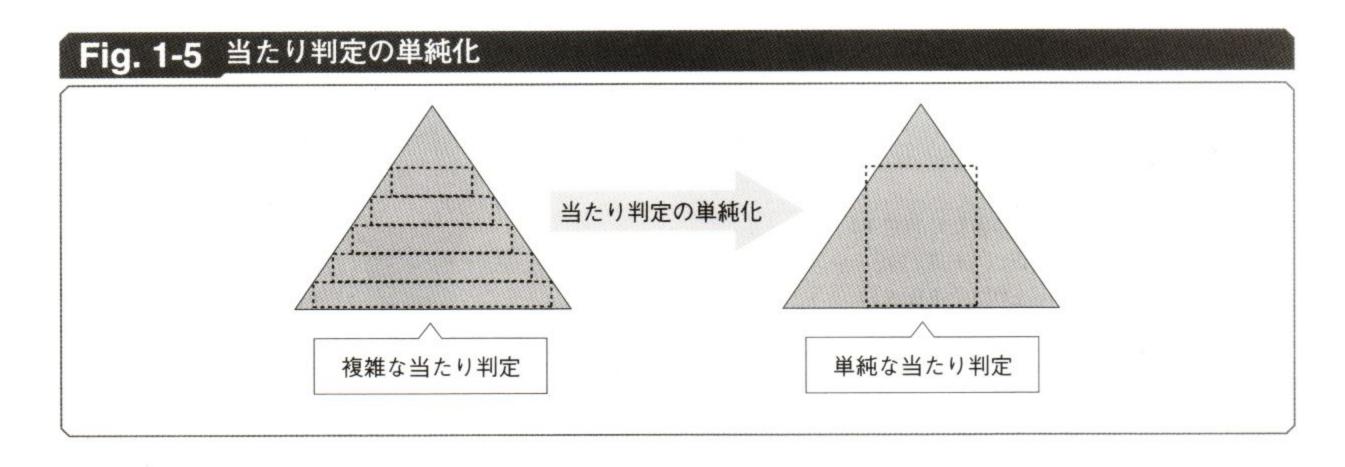
2Dのシューティングゲームにおける当たり判定処理は、それほど重いものではありません。当たり判定処理を躍起になって高速化する必要はないでしょう。むしろ他の方面、例えば3Dグラフィックなどを利用している場合には描画処理などを改善した方が、高速化の効果が大きいでしょう。

とはいえ、下記のような当たり判定処理を軽量化する工夫は有効です。

## 当たり判定の単純化

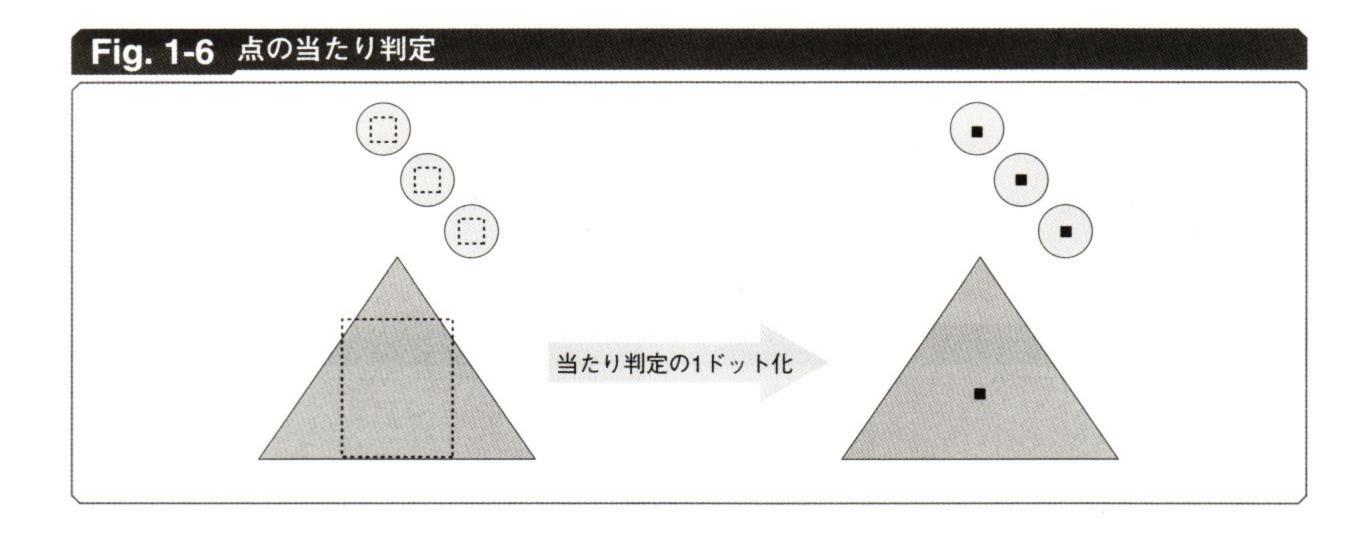
多数の矩形を使えば、当たり判定の形をキャラクターの形にぴったりと合わせることができますが、当たり判定処理の負荷はそれだけ重くなります。そのため、実際のシューティングゲームでは、かなり大胆に単純化した当たり判定を用いることが多いようです。

複雑な形のキャラクターに複数の矩形を使うことはやむをえません。しかし、単純な形のキャラクターはなるべく1個の矩形ですませるようにすると、当たり判定処理を高速化することができます（Fig. 1-5）。

**Fig. 1-5** 当たり判定の単純化

## 当たり判定の1ドット化

小さなキャラクターの当たり判定は矩形にするのではなく、1ドット、つまり点にしてしまうのも高速化のためには有効です（Fig. 1-6）。点と点の当たり判定処理は、矩形と矩形の当たり判定処理よりも条件式が単純なので、高速に処理できます。

例えば、弾の座標を（BX, BY）、自機の座標を（SX, SY）とすると、当たり判定処理の条件式は次のようになります。これは点と点の場合です。

Fig. 1-6 点の当たり判定

```
BX==SX && BY==SY
```

いわゆる弾幕系と呼ばれるシューティングゲームでは、画面上に膨大な数の弾が出現します。この弾と自機の接触を調べるときに、弾と自機の当たり判定を1ドットにしておけば、高速に当たり判定処理が行えます。ただし、この方法が使えるのは、自機の当たり判定が非常に小さくてもかまわないゲームの場合だけです。

弾と自機の片方、例えば弾の当たり判定だけ1ドットにするのはどうでしょうか。実は、このような点と矩形の当たり判定処理は、矩形と矩形の当たり判定処理に比べて、条件式が簡単になりません。

例えば、弾の座標を (BX, BY)、自機の当たり判定を (SL, ST) ～ (SR, SB) とすると、条件式は以下のようになります。これは点と矩形の場合です。

```
SL<=BX && BX<SR && ST<=BY && BY<SB
```

弾の当たり判定が (BL, BT) ～ (BR, BB) の場合、条件式は以下のようになります。これは矩形と矩形の場合です。

```
BL<SR && SL<BR && BT<SB && ST<BB
```

このように、点と点の場合は条件式が簡単になりますが、点と矩形の場合には、矩形と矩形の場合に比べて簡単にはならないことを憶えておきましょう。

## バウンディングボックス

複雑な形状のキャラクターに対する当たり判定処理を行う場合には、事前に簡略な当た

り判定処理を行い、必要な場合だけ複雑な当たり判定処理を行うようにするのが有効です。

例えば、多数の矩形から構成されている複雑な当たり判定があるときには、すべての矩形を囲むような大きな矩形の当たり判定を外側に用意します。これを「バウンディングボックス」と呼びます(Fig. 1-7)。

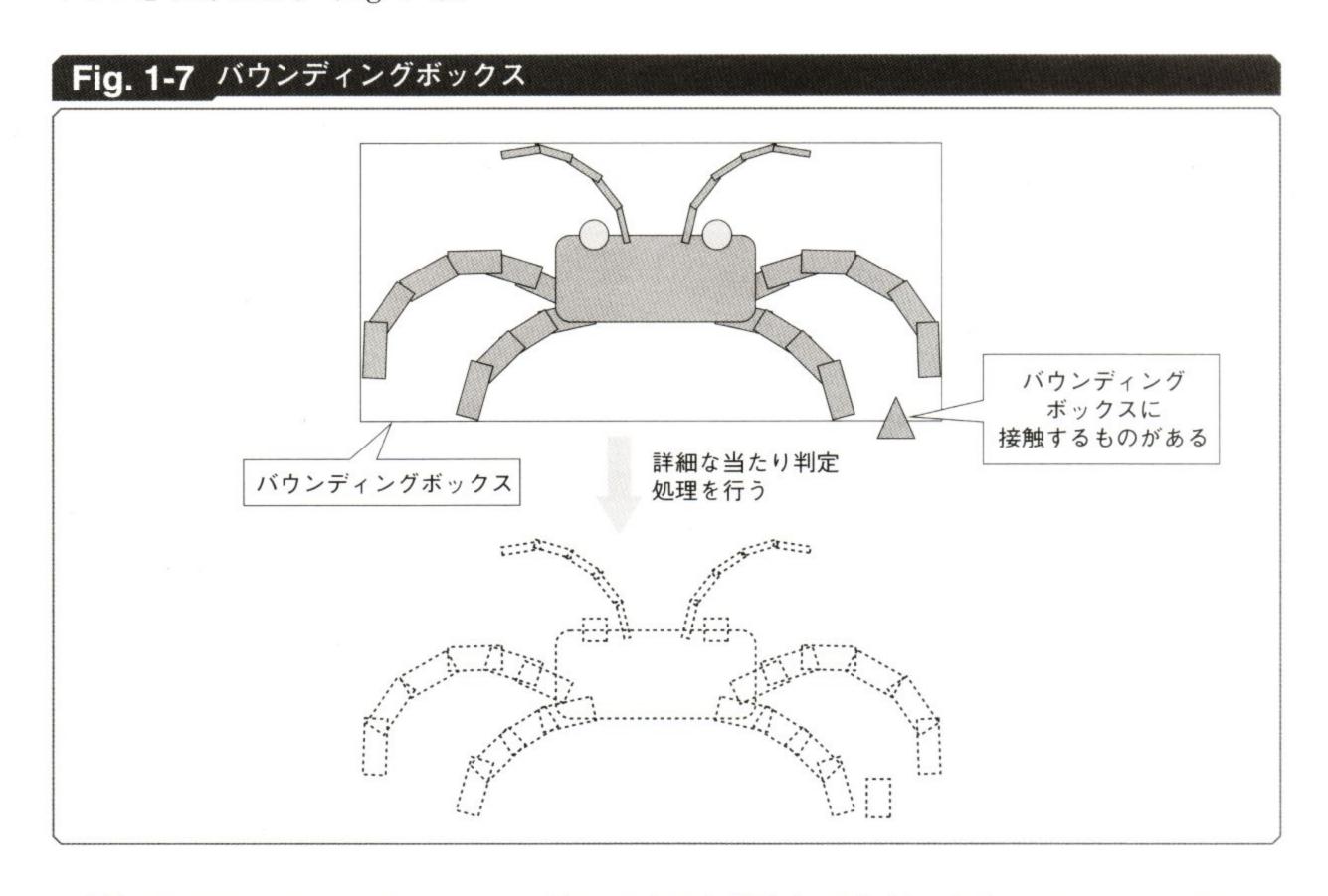



**Fig. 1-7 バウンディングボックス**

最初はバウンディングボックスに対して当たり判定処理を行います。そして、バウンディングボックスに接触したときだけ、内側にある複雑な当たり判定に対して当たり判定処理を行います。ほとんどの場合は、バウンディングボックスに対する当たり判定処理だけで用がすむため、複雑な当たり判定を使っていても高速に処理できることが、この技法の利点です。

バウンディングボックスの技法は3Dゲームでもよく使われています。より一般的には、「バウンディングボリューム」と呼びます。バウンディングボリュームは、3Dオブジェクトをすっぽり取り囲む、直方体や球といった単純な形状のことです。3Dでは2Dよりも当たり判定処理の負荷が重いため、最初にバウンディングボリュームに対する大まかな当たり判定処理を行い、接触したときだけ詳細な当たり判定処理を行うことによって、処理を大幅に高速化できます。

# 特定の敵を特定のタイミングで出現させるには

一般にシューティングゲームには、5ステージや8ステージといった複数のステージがあります。そして、多少のランダム要素は入るものの、基本的に各ステージでは決まった敵が決まったタイミングで出現します。

このように特定の敵を特定のタイミングで出現させるには、敵の種類と出現時刻をテーブルにしておきます（Fig. 1-8）。必要に応じて、敵の出現位置などもいっしょに格納します。

出現時刻というのは、ステージ開始後からの経過時間です。時間の単位は1/100秒でも1/1000秒でもかまいません。ゲームの進行を画面の更新周期（1/60秒など）に合わせる場合には、時間の単位もこの周期に合わせると簡単でしょう。

プログラムは経過時間を監視して、敵の出現時刻になったら、テーブルを参照しながら指定された種類の敵を指定された座標に出現させます。ステージ別にテーブルを用意しておけば、敵の出現シーケンスが異なるステージをいくつでも提供することができるというわけです。

テーブルは配列としてプログラム内に保持してもかまいませんし、外部のデータファイルにする方法もあります。前者の利点はプログラムが簡単になることで、後者の利点はプログラマー以外もデータを編集できるということです。本書ではテーブルを外部のデータファイルにする方法を、Chapter 8で解説します（P. 256）。

**Fig. 1-8** 敵の種類・座標・出現時刻のテーブル

| 敵の種類 | 出現位置 | 出現時刻 |
|---|---|---|
| ENEMY_A | (10, 20) | 0 |
| ENEMY_B | (30, 40) | 0 |
| ENEMY_A | (50, 10) | 15 |
| ENEMY_C | (20, 30) | 25 |

⋮

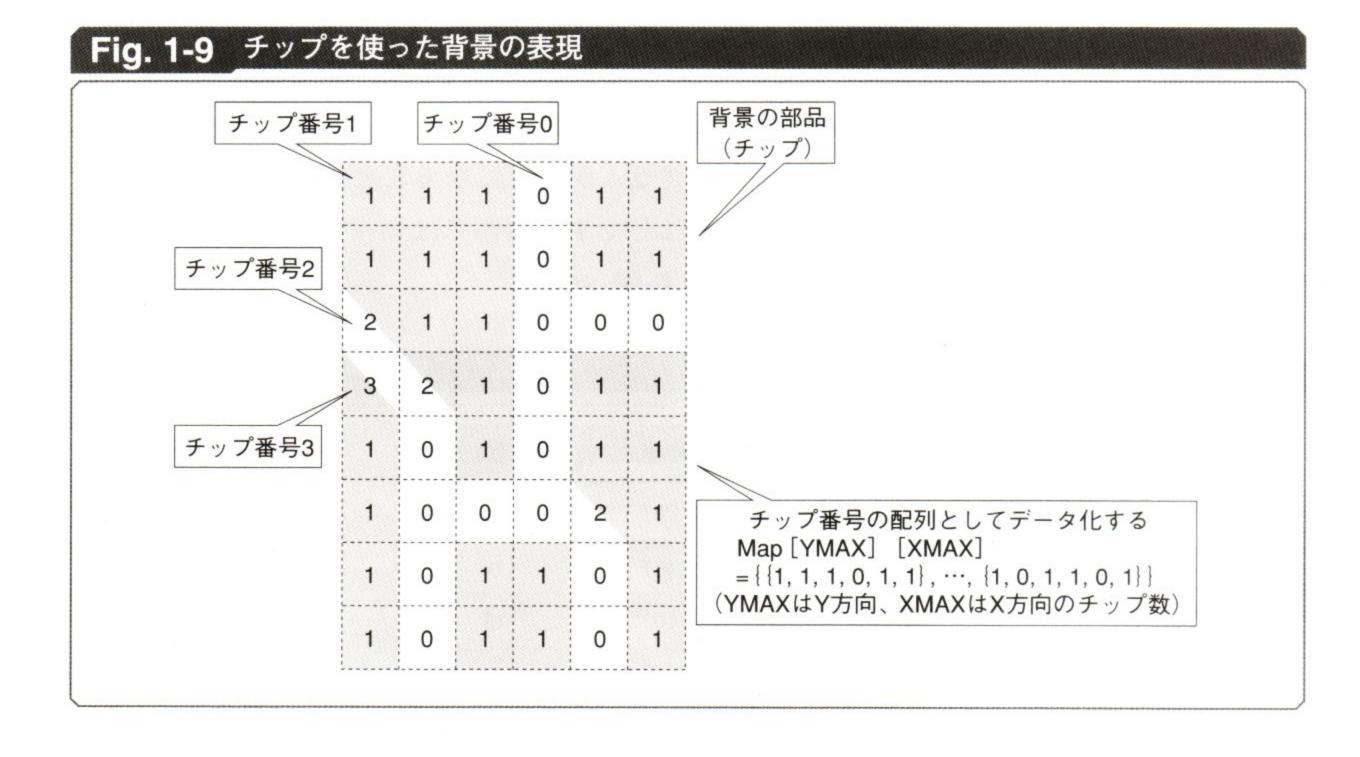
# 背景はどうやって動かすのか

多くのシューティングゲームは自機や敵の背後に背景を表示します。そして、背景を一定方向にスクロールさせます。

ほとんどのトップビュー（上空から見た視点）のゲームは上から下にスクロールします。サイドビュー（真横から見た視点）のゲームでは、右から左にスクロールするのが一般的です。

背景を表現する代表的な手段としては、チップを並べる方法があります（Fig. 1-9）。チップは背景の部品となる小さな矩形の画像です。配置するチップの番号を配列としてデータ化することにより、背景を構成します。

2Dグラフィックのシューティングゲームではチップを使うのが一般的です。一方、最近では3Dグラフィックを使って背景を作るシューティングゲームも増えてきました。本書では3Dグラフィックを使った背景の作り方を、Chapter 8で解説します（P. 238）。

**Fig. 1-9　チップを使った背景の表現**

## サンプルシューティングゲーム「紫雨」の紹介

本書では実際にゲームを制作しながら、シューティングゲームの制作方法を解説します。ここで、本書で作成するサンプルゲーム「紫雨」(むらさめ)について紹介します(Fig. 1-10、11)。

ご覧のとおり、紫雨は寿司をモチーフにしたシューティングゲームです。もちろん、ネタが寿司だというだけで、ゲーム性やプログラムの構造などはオーソドックスなシューティングゲームです。この紫雨を参考に、お好みのグラフィックやサウンドを使って、オリジナルのゲームを作り上げていただければと思います。グラフィックやサウンドなどをア

**Fig. 1-10** 紫雨のタイトル画面

**Fig. 1-11** 紫雨のゲーム画面

レンジしてオリジナルのゲームを作るためのヒントは、Chapter 12で解説します（P. 353）。
紫雨は実際に遊ぶことができます。付録CD-ROMにソースコードおよび実行ファイルの一式を収録したので、ぜひ実際に動かしながら本書をお楽しみください。

## サンプルの実行方法

本書の付録CD-ROMには、紫雨をはじめとした多くのサンプルプログラムが収録されています。これらのサンプルを実行するには、付録CD-ROMのexeファイルをエクスプローラから実行してください。各サンプルの内容は、本書の対応する章で紹介しています。
サンプルを実行する場合には、事前に下記のソフトウェアをインストールしておく必要があります。

**DirectX 9.0c ランタイム**
http://www.microsoft.com/japan/windows/directx/downloads/default.mspx

**DirectX End User Runtimes (August 2006)**
http://www.microsoft.com/japan/msdn/directx/downloads.aspx

## 紫雨の遊び方

紫雨の完成版は付録CD-ROMの「ShtGame_Arrange」フォルダに収録されています。実行ファイルは「ShtGame_Arrange¥Release¥ShtGame.exe」です。紫雨を起動するには、エクスプローラからこのexeファイルを実行してください。
ゲームの操作はキーボードまたはジョイスティックで行います。主な操作は以下のとおりです。

- カーソルキーまたはスティックの上下左右で自機の移動
- 「Z」キーまたはボタン「0」でゲームスタート
- 「Z」キーまたはボタン「0」を軽く連打でショット
- 「Z」キーまたはボタン「0」を押しっぱなしでビーム
- 「X」キーまたはボタン「1」でボム
- 「C」キーまたはボタン「2」でポーズおよびコンティニュー
- 「Esc」キーで終了

ショットを押しっぱなしにするとビームが出ます。ビームはショットに比べると攻撃範囲が狭いのですが、威力は高めになっています。また、ビームを押している間は自機の移

動が遅くなります。これは細かい動きで弾幕を抜けるときに便利です。

なお、ショット発射中のボムは弾を消すノーマルボム（またはアトラクトボム）、ビーム発射中のボムは弾と敵の動きを遅くするスローボム（またはストップボム）になります。ボムのタイプはゲーム開始時に選択することができます。また、敵や弾を切り裂くソードやワイパーという特殊な武器も選択可能です。

紫雨の起動時オプションに「-w 1024 -h 768」などを指定すると、解像度を変更することができます。フルスクリーン時の解像度は「-fw 1280 -fh 1024」のように指定します。

## 開発環境の準備

本書では開発環境としてVisual C++とDirectXを使います。付録CD-ROMに収録されたサンプルのプロジェクトをビルドする場合には、事前に下記のソフトウェアをインストールしておく必要があります。サンプルを実行するだけならば、これらのソフトウェアは必要ありません。

### Visual C++

本書のサンプルは、Visual C++ .NET 2003（Visual Studio .NET 2003）で作成されています。サンプルは、Visual C++ 2005にも対応しています。

付録CD-ROMにはVisual C++ .NET 2003用とVisual C++ 2005用、両方のプロジェクトファイルが収録されています。

Visual C++ 2005 Express Editionをご使用の場合は、Appendix（P. 380）をご参照ください。Express EditionでWin32アプリケーションを作成する場合は、手作業による設定が必要になります。Appendixに、その手順と注意点をまとめました。

### DirectX SDK（August 2006）

DirectX 9.0が必要です。本書のサンプルは、2006年8月にリリースされたアップデート（August 2006）に対応しています。下記のWebサイトからSDKをダウンロードして、システムにインストールしてください。

**DirectX SDK（August 2006）**
http://www.microsoft.com/japan/msdn/directx/downloads.aspx

### Platform SDK

Visual C++ 2005 (Visual Studio 2005) で本書のサンプルのプロジェクトをビルドするためには、Platform SDKが必要となります。Microsoftのダウンロードセンターから、「Windows Server 2003 SP1 Platform SDK」をダウンロードし、インストールを行ってください。なお、インストール後の設定などは、Appendix (P. 380) を参考にしてください。

**ダウンロードセンター**

http://www.microsoft.com/downloads/

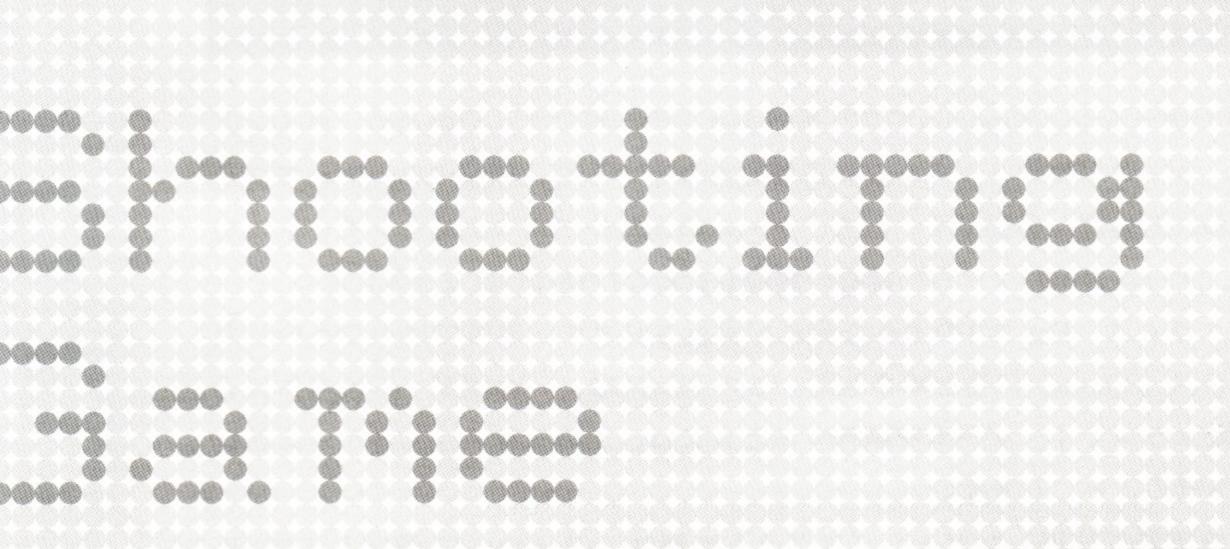
## >>Chapter 1のまとめ

本章ではシューティングゲームの全体像を確認し、プログラミングの要点、開発環境の準備について整理しました。まずは、どんなゲームを作りたいのかイメージをふくらませながら、サンプルを動かしてみてください。開発環境を用意しておけば、次章以降、気になる部分を自分なりにアレンジして動かしてみることもできます。

次章では、DirectXを使ったプログラミングの基本、グラフィックの表示方法、入力の取得方法、サウンドの再生方法などについて解説します。

# Chapter 02

# ゲームライブラリ

本章では、DirectXを簡単に使うためのゲームライブラリを構築します。DirectXのなかからシューティングゲームの制作に必要な機能を選び出して、これらの機能を使いやすいクラスに整理します。
この章は順にじっくり読むというよりも、どんな機能があるのかをざっと眺めていただければ十分です。そして後の章を読み進める際に、必要に応じてリファレンス的に利用していただくと便利です。

## ゲームライブラリを作る

Windowsでゲームを制作するときに使う最も代表的なライブラリといえば、DirectXです。DirectXを使うと高速で派手なゲームを作ることができます。ただ、DirectXを使いこなすには多少の慣れが必要なので、初めて使うときには難しく感じてしまうかもしれません。

そこで本書では、DirectXを簡単に使うために、オリジナルのゲームライブラリを作ることにしました。DirectXのなかからシューティングゲームの制作に必要な機能を選び出して、これらの機能を使いやすく整理します。

本章では、このゲームライブラリの内部構造について解説しますが、最初は軽く読み流していただいて大丈夫です。もし後の章を読み進めるうちに気になるところが出てきたら、適宜本章を読み返してみてください。

ここで制作したゲームライブラリの内部構造は多少複雑ですが、ライブラリを使うだけならば簡単です。大まかにどんな機能があるのかだけを知っておき、こういったゲームライブラリを自作する際に詳細を読んでいただければと思います。

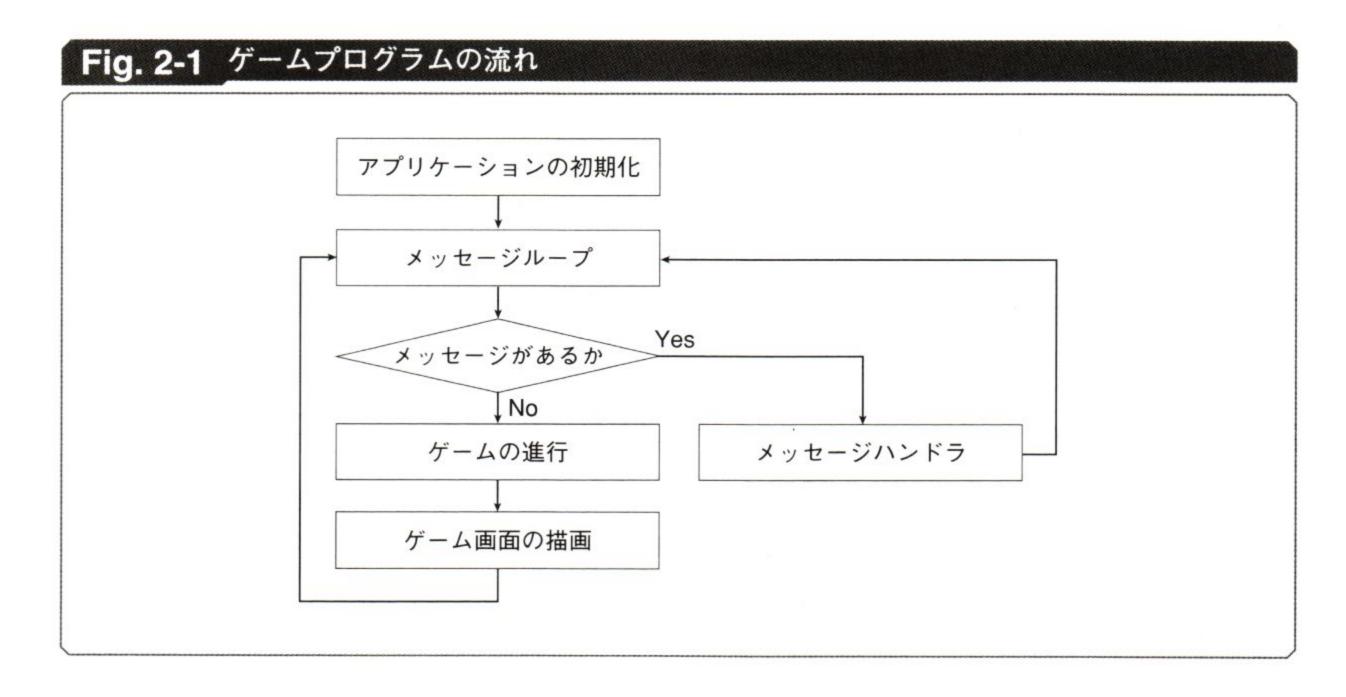
なお、ゲームライブラリのプロジェクトは、付録CD-ROMの「LibGame」フォルダに収録しています。本文ではプログラムの仕組みや大まかな処理の流れを解説していますので、実際のソースコードと適宜あわせてご覧いただくと、より理解が進むと思います。

## ゲーム本体の機能

DirectXを使ったゲームも、Windowsアプリケーションの一種です。そのため、ウィンドウを作成したり、ウィンドウのメッセージに応答したりといった、通常のWindowsアプリケーションで行われる処理が必要です。一方、ゲームの進行やフルスクリーンモードの切り替えといった、ゲームのプログラムに特有の処理も行わなければいけません。

ゲームのプログラムは、Fig. 2-1のような流れで実行されます。本書では、こういった処理をゲームクラス（CGame）にまとめました。このクラスは、いわばゲームの本体に必要な機能を集めたクラスです。ゲームを制作するときには、このクラスから各ゲーム独自のクラスを派生させることにします。

ゲームクラスを実際に使用したゲームプログラムの例は、Chapter 4で解説します（P. 105）。

**Fig. 2-1** ゲームプログラムの流れ

## アプリケーションの初期化

ゲームクラス（CGame）のコンストラクタで、アプリケーション全体の初期化を行います。ここで行うのは次のような作業です。

- ウィンドウクラスの登録
- アクセラレータキーの設定
- ウィンドウの作成
- ウィンドウハンドルとインスタンスの対応づけ
- グラフィックの初期化
- 時間の初期化

### ウィンドウクラスの登録

ウィンドウを作る前に、ウィンドウクラスを登録する必要があります。ウィンドウクラスというのは、ウィンドウを作成するために必要な情報を保持する一種のオブジェクトです。ウィンドウクラスには、アプリケーション名、アイコン、メッセージハンドラへの関数ポインタといった情報を登録します。

アイコンはウィンドウの左上に表示されます。リソースなどから画像ファイル（アイコンファイル）を読み込むようにするとよいでしょう。

メッセージハンドラは、ウィンドウのメッセージに応答する関数です。この段階では、何も行わないダミーのメッセージハンドラを登録しておきます。初期化処理をひととおり終えたら、正式なメッセージハンドラに入れ替えます。ダミーのメッセージハンドラを使

うのは、初期化が完了していない状態で正式のメッセージハンドラが呼び出されると、未初期化の機能が使用される危険性があるためです。

### アクセラレータキーの設定

アクセラレータキーとは、そのキーを押すとウィンドウに特定のメッセージが送信されるキーのことです。例えば、ゲームの終了や画面モードの切り替えのためのキーを、アクセラレータキーとして登録します。

### ウィンドウの作成

ウィンドウを作成します。この際には、ウィンドウのクライアント領域（ウィンドウ内部の領域）を指定されたサイズにするため、前もってウィンドウの外枠の大きさを計算しておきます。

### ウィンドウハンドルとインスタンスの対応づけ

ウィンドウハンドルというのは、ウィンドウを識別する一種の番号です。詳しくは後述しますが、ウィンドウのメッセージをゲームクラス（CGame）で処理するためには、ウィンドウハンドルとゲームクラスのインスタンスの対応を保存しておく必要があります。さもないと、ウィンドウにメッセージが送信されたときに、そのメッセージをゲームクラスのインスタンスで処理することができません。

### グラフィックの初期化

Direct3Dを初期化します。Direct3Dデバイスの作成、画面モードの設定、画面のクリアなどを行います。

### 入力の初期化

キーボードやジョイスティックからの入力を読み取るための準備です。DirectInputを初期化します。

### 時間の初期化

時間（タイマー）に関する変数を初期化します。これらの変数は、一定時間間隔でゲームを進行させるために使います。ゲームクラス（CGame）では、VSYNC（垂直同期期間）に合わせてゲームを進行させる方法と、タイマーに合わせてゲームを進行させる方法の、両方をサポートしています。

## ウィンドウに送られたメッセージを処理する

メッセージハンドラは、ウィンドウに送信されたメッセージを処理する関数です。ほとんどのWindowsアプリケーションにはメッセージハンドラがあります。メッセージハンドラでは、次のようなメッセージを処理します。

- 描画（WM_PAINTメッセージ）
- タイトルバーのダブルクリック（WM_NCLBUTTONDBLCLKメッセージ）
- ウィンドウを閉じる（WM_CLOSEメッセージ）
- コマンド（WM_COMMANDメッセージ）
- システムコマンド（WM_SYSCOMMANDメッセージ）
- ウィンドウの破棄（WM_DESTROYメッセージ）

### 描画（WM_PAINTメッセージ）

アプリケーションの画面を描画するように要請されたときに、ゲーム画面を描画します。

### タイトルバーのダブルクリック（WM_NCLBUTTONDBLCLKメッセージ）

タイトルバーをダブルクリックしたときには、ウィンドウモードとフルスクリーンモードを切り替えます。

### ウィンドウを閉じる（WM_CLOSEメッセージ）

ウィンドウを閉じるよう要請されたときには、終了確認のダイアログを表示してから、ウィンドウを破棄します。

### コマンド（WM_COMMANDメッセージ）

各種のコマンドに関するメッセージ処理です。例えば、終了キー（「Esc」キー）を押したときや、画面モード切り替えキー（「Alt」+「Enter」キー）を押したときの処理を行います。

### システムコマンド（WM_SYSCOMMANDメッセージ）

システムコマンドとは、Windowsのシステムに関するさまざまなメッセージです。例えば、ウィンドウを最大化したときの処理などを行います。

また、スクリーンセーバーとモニターの電源切断を防止するための処理も行うとよいでしょう。これは、ジョイスティックを使うゲームで、キーボードやマウスからまったく入

力がない状態で長時間遊ぶと、スクリーンセーバーが動いてしまったり、モニターの電源が切れてしまったりすることがあるからです。

### ウィンドウの破棄（WM_DESTROYメッセージ）

ウィンドウを破棄するよう要請されたら、プログラムを終了します。

※

本書では2種類のメッセージハンドラを用意します。1つは非メンバ関数、もう1つはゲームクラス（CGame）のメンバ関数です。

メッセージを実際に処理するのは、メンバ関数のメッセージハンドラです。しかし、Win32 APIの仕様上、ウィンドウに対するメッセージを直接受け取ることができるのは、非メンバ関数だけです。そこで、非メンバ関数のメッセージハンドラを中継役にして、メンバ関数のメッセージハンドラにメッセージを転送します。

アプリケーションを初期化する際に、ウィンドウハンドルに対応するゲームクラスのインスタンスを保存しておきました（P. 22）。非メンバ関数のメッセージハンドラは、この情報を用いて、どのインスタンスにメッセージを転送するべきかを判断します。

## メッセージループ

Windowsアプリケーションのメインループに相当するのがメッセージループです。メッセージループでは、ウィンドウに送信されたメッセージを取り出し、対応するメッセージハンドラを呼び出すことを繰り返します。

一般のアプリケーションではこの繰り返しだけなのですが、ゲームの場合には少し違います。メッセージを処理する一方で、処理すべきメッセージがないときには、ゲームの処理を進行します。

メッセージループに入る前には、グラフィックの初期設定を行います。設定内容はゲームによって違うので、仮想関数などを用いて、ゲームごとに異なる初期設定処理を記述できるようにしておくとよいでしょう。

また、メッセージハンドラの置き換えも行います。ウィンドウ作成時にはダミーのメッセージハンドラを登録しましたが（P. 21）、これを正式なメッセージハンドラに置き換えます。

続いて、メッセージループに入ります。メッセージループでは、プログラムの終了を表すメッセージ（WM_QUITメッセージ）を受け取るまで、次のような処理を繰り返します。

- バックグラウンド時のメッセージ処理
- フォアグラウンド時のメッセージ処理

・時間調整
・移動と描画

### バックグラウンド時のメッセージ処理

アプリケーションがアクティブでないときには、Win32 APIのGetMessage関数を使って、ウィンドウに送られたメッセージを処理します。これは、一般的なWindowsアプリケーションと同様のメッセージ処理です。

### フォアグラウンド時のメッセージ処理

アプリケーションがアクティブなときには、Win32 APIのPeekMessage関数を使って、ウィンドウに送られたメッセージを処理します。GetMessage関数はウィンドウが次のメッセージを受け取るまで待機しますが、PeekMessage関数はメッセージの有無にかかわらず待機しません。この性質を利用して、メッセージがないときにはゲームの進行に関する処理を行います。

GetMessage関数とPeekMessage関数を使い分けずに、常にPeekMessage関数だけを使うこともできます。しかし、この場合はWindowsのシステムに制御がなかなか戻らないため、他のアプリケーションの反応が悪くなります。そこで、ゲームアプリケーションがアクティブのときだけPeekMessage関数を使い、アクティブでないときにはGetMessage関数を使うようにします。

### 時間調整

時間を計測し、前回計測時からの経過時間に基づいてゲームの進行速度を調整します。精度の高い時間計測を行うためには、Win32 APIのQueryPerformanceFrequency関数とQueryPerformanceCounter関数を使うとよいでしょう。これらの関数が使えない環境では、かわりにWin32 APIのGetTickCount関数を使います。GetTickCount関数の精度は1/1000秒です。

### 移動と描画

ゲームの進行（移動）と、ゲーム画面の描画を行います。具体的な移動処理や描画処理については、仮想関数をオーバーライドすることによって、ゲームごとに異なる処理を記述します。

なお、ここではフレーム落ちまたは処理落ちのどちらかを使って、ゲームの速度調整を行うことができます（P. 207）。

※

終了メッセージ（WM_QUIT）を受け取ったら、プログラムを終了します。メッセージハ

ンドラをダミーに戻し、ウィンドウのクラスを破棄します。メッセージハンドラをダミーに戻すのは、各種のオブジェクトを破棄した後に、それらのオブジェクトを使う処理が誤って呼び出されるのを防ぐためです。

## 処理落ちとフレーム落ち

画面に登場するキャラクターを増やしたり、エフェクトを追加したりすると、だんだんゲームの処理は重くなっていきます。本書のサンプルの場合、通常は1/60秒間隔でキャラクターの移動と描画を行いますが、処理が重くなると1/60秒間隔では間に合わなくなる（処理しきれなくなる）ことがあります。処理が間に合わないと、一時的にスローモーションがかかったような動きになります。これを一般に処理落ちと呼びます（Fig. 2-2）。

市販のゲームでも処理落ちはよく見かけます。ただし、ターゲットとなるハードウェアが限定されている場合、処理落ちを演出として利用することも可能です。PC向けゲームの場合には、使用するPCのスペックによって処理落ちの発生状況が変わってしまうことに注意が必要です。処理落ちが発生するとゲームの進行が遅くなるため、ゲームが簡単になります。そのため、速いPCと遅いPCとではゲームの難易度が変わってしまうのです。

処理落ちを避けるには、遅いPCでも処理落ちが起きないように処理を軽くするのがベストの方法ですが、フレーム落ち（コマ落ち）を使って速度を調整する方法もあります。これは、処理が間に合わない場合に描画を省くことによって、速度の低下を緩和する方法です。

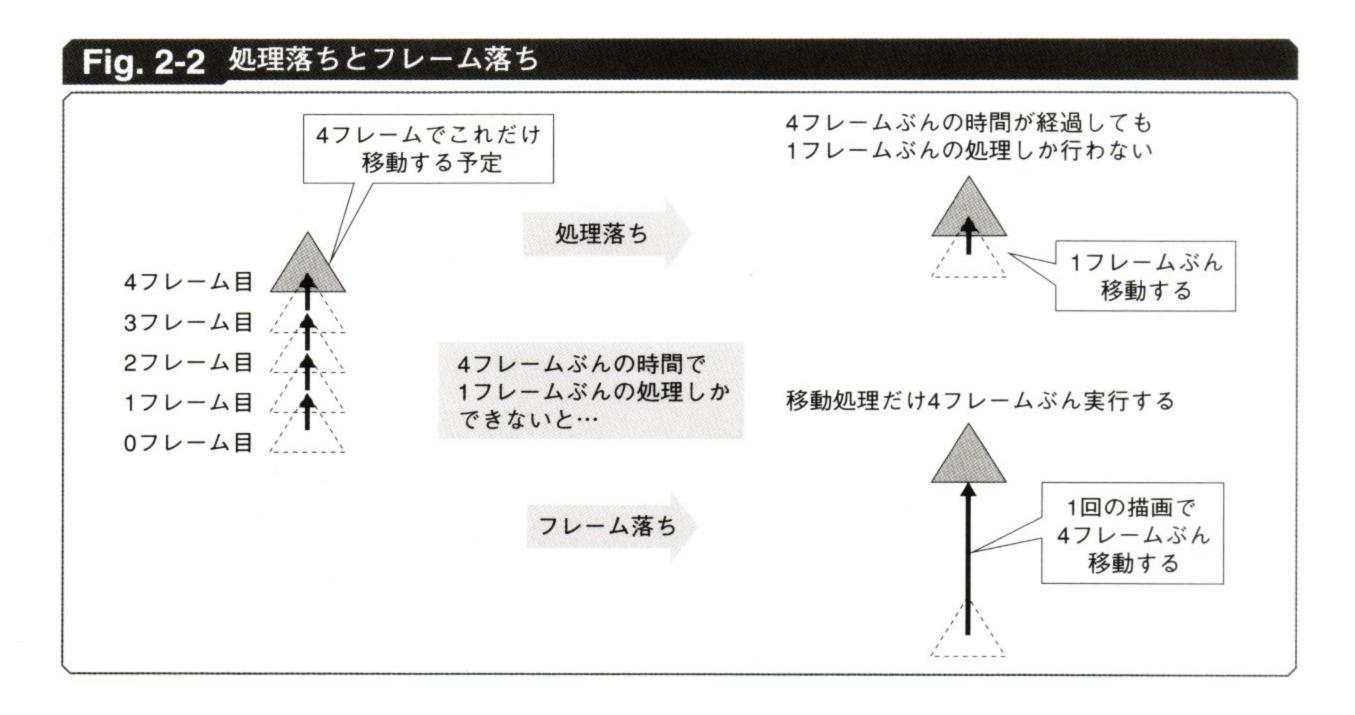



**Fig. 2-2** 処理落ちとフレーム落ち

フレーム落ちを使うと、ゲームの進行速度をほぼ一定に保つことができます。ただ、処理が間に合わない場合にはアニメーションがなめらかではなくなってしまうので、できるかぎりフレーム落ちが起きないように処理を軽くすることも大事です。

ゲームクラス（CGame）は、処理落ちとフレーム落ちの両方に対応しています。用途に応じて、どちらを使うのかを切り替えることができます。

フレーム落ちを制御するには、前回計測時からの経過時間を計測します。そして、この経過時間を移動に使用できる時間として扱います。

具体的には、時間経過を1フレームに相当する時間で割った回数だけ移動処理を行い、最後に1回だけ描画処理を行います。前回計測時からの経過時間が長いとき、つまり処理が重いときには、移動処理を複数回行うわけです。割り算で余った経過時間の端数は保存しておき、次のフレームで使用します。

一方、処理落ちの場合には、常に移動処理と描画処理を1回ずつ実行します。経過時間は考慮しません。

いずれの場合も、画面を描画する処理のなかで、VSYNC（画面の垂直同期期間）を待ちます。ゲームが進行する周期はVSYNCによって制限されるため、進行が速くなりすぎるということはありません。多くのゲームでは、VSYNCの頻度は秒間60回です。

## デバイスロストとデバイスリセット

DirectXでグラフィックを扱うときには、デバイスロストとデバイスリセットに関して知っておく必要があります。ここでは、画面の描画や画面モードの切り替えについて説明するのに先だって、デバイスロストとデバイスリセットに関して解説します。

デバイスロストは、グラフィックのデバイスが一時的にプログラムから使えない状態になることです。これは画面モードを切り替えたときなどに起こります。一方のデバイスリセットは、デバイスロストの状態から復帰して、再びデバイスが使える状態にする処理のことです。

デバイスロストがいつ起こるのかはわかりません。ゲーム専用機とは違って、Windows環境では複数のアプリケーションが同時に走っているのが普通です。フルスクリーンモードでゲームが動いていても、ユーザーが急に他のアプリケーションをアクティブにするかもしれません。

そのため、ゲームの画面を描画するときには、デバイスロストが起きていないかどうかを調べる必要があります。もし起きていたらデバイスリセットを実行して、再びデバイスが使える状態にします。

## 画面の描画

ゲーム画面を描画する際には、まず最初にデバイスロストが起きたかどうかを調べます。そして、起きていたらデバイスリセットを行います。デバイスロストの有無を調べるには、IDirect3DDevice9::TestCooperativeLevel関数を使います。

関数の戻り値がD3DERR_DEVICELOSTのときには、デバイスロストが起きていて、かつすぐには復帰できない状態です。この場合は何もせずに、デバイスリセットが可能になるのを待ちます。

関数の戻り値がD3DERR_DEVICENOTRESETのときには、デバイスロストが起きていて、かつデバイスリセットが可能な状態です。この場合はデバイスリセットを行います。

デバイスリセットを行う前と、デバイスリセットを行った後には、ゲームによって異なる設定処理が必要なことがあります。例えば、仮想関数を使って、ゲームに応じた設定処理を行えるような仕組みを作っておくとよいでしょう。

次に、描画を開始します。Direct3Dでは、描画の開始時にIDirect3DDevice9::BeginScene関数を呼び出す必要があります。

ゲーム画面の描画処理はゲームによって異なります。例えば仮想関数を使って、ゲームごとに異なる描画処理を記述できるような仕組みにするとよいでしょう。

描画が終わったら、IDirect3DDevice9::EndScene関数を呼び出します。そして、描画結果を画面に表示するために、IDirect3DDevice9::Present関数を呼び出します。

## 画面モードの切り替え

フルスクリーンモードとウィンドウモードを切り替えるには、まず画面モードに応じたパラメータを設定します。パラメータは画面のサイズやリフレッシュレートなどです。そしてデバイスをリセットすると、画面モードが切り替わります。

なお、ウィンドウモードではアプリケーションが自由にリフレッシュレートを設定することができません。Direct3Dの場合には、リフレッシュレートに0を指定すると、既定のリフレッシュレートが使用されます。

※

ゲーム本体の機能は「Game.h」と「Game.cpp」にまとめました。詳しくは、付録CD-ROMの「LibGame￥Game.h」と「LibGame￥Game.cpp」をご参照ください。

# グラフィックの基本機能

次はグラフィッククラス（CGraphics）について解説します。これは、グラフィックまわりの基本的な処理をまとめたクラスです。このクラスを実際に使用したゲームプログラムの例は、Chapter 4で解説します（P. 105）。

## グラフィックの初期化

グラフィック機能を初期化する際には、最初にウィンドウスタイルやクライアント領域のサイズといった、ウィンドウの情報を保存します。これらは画面モードを変更するときに使います。

次に、Direct3Dインタフェース（IDirect3D9インスタンス）を作成します。続いて、このDirect3Dインタフェースを使って、Direct3Dデバイス（IDirect3DDevice9インスタンス）を作成します。

## デバイスのリセット

デバイスのリセットと作成は処理がよく似ています。1つの関数に処理をまとめて、両方の処理を行うとよいでしょう。

最初に、ウィンドウのスタイルとサイズを設定します。デバイスのリセットや作成の前にウィンドウの設定を行わないと、ウィンドウのサイズが狂ってしまうことがあります。

次に、デバイスの作成とリセットに必要なパラメータを設定します。設定項目は画面のサイズ・画面モード・リフレッシュレートなどです。

パラメータを設定したら、デバイスをまだ作っていないときには作成し、すでに作ってあるときにはリセットします。デバイスを作成する場合には、条件を変えながらIDirect3D::CreateDevice関数を複数回呼び出します。これは、なるべく速度面や機能面で有利なデバイスを選ぶためです。

リセットや作成に成功したら、デバイスに関する情報を取得します。画面のサイズやリフレッシュレートなどを取得しておくと便利です。

※

グラフィックまわりの基本機能は「Graphics.h」と「Graphics.cpp」にまとめました。詳しくは、付録CD-ROMの「LibGame¥Graphics.h」と「LibGame¥Graphics.cpp」をご参照ください。

# 3Dモデルの読み込みと描画

続いては、3Dモデルの読み込みと描画を行うメッシュクラス（CMesh）について紹介します。このクラスは、DirectXのID3DXMeshクラスをベースにしています。Xファイル形式（拡張子.x）の3Dモデルを読み込んで、画面に描画することができます。本書のサンプルは、自機などのキャラクターに、Xファイル形式の3Dモデルを用いています。

市販ゲームなどでは専用の3Dフォーマットを使うことがありますが、ホビーで作るゲームならば、手軽に使えるXファイルがお勧めです。今ではほとんどの3Dツールで、3DモデルをXファイルとして出力することができます。

3Dツールには高価なものが多いですが、低価格で使いやすいものもあります。以下に低価格なツールをいくつか紹介します。

## Metasequoia

本書の3Dモデルは、ほとんどこのMetasequoia（メタセコイア）のシェアウェア版を使って制作しました。Metasequoiaは低価格（本書の執筆時点（2006年8月）では5,000円）ながらも本格的な3Dツールです。フリーウェア版もあるので、最初はこちらを使い、気に入ったらシェアウェア版を使ってみてはいかがでしょうか。

Metasequoia
http://www.metaseq.net/

## gameSpace Light

trueSpaceという3Dツールで有名なCaligari社の製品です。ポリゴン数に上限が設けられているものの、無償で利用することができ、なおかつアニメーションにも対応していることが魅力です。ポリゴン数の制限がない有償版もあります。

gameSpace Light
http://www.caligari.com/gamespace/

メッシュクラスを実際に使用したゲームプログラムの例は、Chapter 4で解説します（P. 105）。本書のサンプルでは、自機・弾・敵などのキャラクターを、すべてメッシュクラスを使って描画しています。

## 3Dモデルの読み込み

3Dモデルを読み込むには、最初にXファイルをロードします。これはD3DXLoadMeshFromX関数を呼び出すだけなので、非常に簡単です。

次にマテリアルを読み込みます。マテリアルは3Dモデルに適用されている色などの属性です。マテリアルは、Xファイルの読み込み時にD3DXLoadMeshFromX関数が読み込んでくれます。このマテリアルはD3DXMATERIAL型ですが、描画の際にはD3DMATERIAL9型の方が使いやすいので、変換しておきます。

続いてテクスチャを読み込みます。テクスチャはXファイルとは別のファイルになっているので、自前で読み込む必要があります。テクスチャのファイル名はXファイルに書かれており、Xファイルの読み込み時にD3DXLoadMeshFromX関数が読み込んでくれます。テクスチャの読み込みにはD3DXCreateTextureFromFile関数を使います。

## 3Dモデルの描画

3Dモデルを描画するときには、同じマテリアルを使うポリゴンごとにグループ分けするのが一般的です。こうすることによって、マテリアルの切り替えに伴うオーバーヘッドを最小にすることができます。同じマテリアルを使うポリゴンのグループを、サブセットと呼びます。

3Dモデルを読み込むときに、マテリアルも読み込みました。このマテリアルをそのまま描画色に使ってもよいのですが、指定した値を色に対して乗算・加算する機能を用意しておくと、3Dモデルの明るさや色調を変えて表示したいときに便利です。

サブセットを描画する際には、まずマテリアルとテクスチャをDirect3Dデバイスに設定します。それぞれ、IDirect3DDevice9::SetMatrial関数とIDirect3DDevice::SetTexture関数を使います。

次に、サブセットを構成するポリゴンを描画します。これは、ID3DXBaseMesh::DrawSubset関数を使います。

List 2-1は、3Dモデルを描画する関数です。これは最も基本的な関数で、この他にサイズや回転などを指定できる関数も用意しています。

※

3Dモデルの読み込みと描画に関する機能は「Mesh.h」と「Mesh.cpp」にまとめました。詳しくは、付録CD-ROMの「LibGame¥Mesh.h」と「LibGame¥Mesh.cpp」をご参照ください。

**List 2-1　3Dモデルの描画（Mesh.cpp）**

```
void CMesh::Draw() {
    D3DMATERIAL9 mat;

    // マテリアルの数だけループする
    for (DWORD i=0; i<NumMaterials; i++) {

        // マテリアルの変更
        // 色に対してColorMultiplierを乗算し
        // ColorAdditionを加算する
        mat=Materials[i];
        D3DXCOLOR* col;
        #define COLOR_OPERATION(TARGET) ¥
            col=(D3DXCOLOR*)&mat.TARGET;¥
            D3DXColorModulate(col, col, &ColorMultiplier);¥
            D3DXColorAdd(col, col, &ColorAddition);
        COLOR_OPERATION(Diffuse);
        COLOR_OPERATION(Ambient);
        COLOR_OPERATION(Specular);
        COLOR_OPERATION(Emissive);

        // 描画色とテクスチャの設定
        Device->SetMaterial(&mat);
        Device->SetTexture(0, Textures[i]);

        // サブセットの描画
        Mesh->DrawSubset(i);
    }
}
```

# 2D画像の読み込みと描画

最近はシューティングゲームでも3Dグラフィックを使うのが当たり前になっていますが、まだまだ2Dグラフィックの需要はあります。特に、多量の弾が出現する弾幕系シューティングゲームなどでは、自機は3Dグラフィックで描いていても、弾の描画には負荷が軽い2Dグラフィックを用いているのをよく見かけます。

DirectXの場合には、2Dグラフィックの描画にもDirect3Dを使います。2Dの画像を表示する場合には、画像をテクスチャとして読み込み、このテクスチャを適用したポリゴンを描画します。

こういった2D画像の読み込みと描画に関する処理を、テクスチャクラス（CTexture）に

まとめました。このクラスは、2D画像をテクスチャとして読み込み、描画することができます。

テクスチャクラスを実際に使用したゲームプログラムの例は、Chapter 6で解説します(P. 185)。本書のサンプルでは、ロゴやタイトル画面の描画にこのクラスを使います。

## 2D画像の読み込み

まず最初に、2D画像をファイルから読み込んで、テクスチャを作成します。これはD3DXCreateTextureFromFileEx関数を呼び出すだけなので、とても簡単です。

次に、テクスチャのサイズと元画像のサイズを取得しておくとよいでしょう。サイズの取得にはIDirect3DTexture9::GetLevelDesc関数を使います。

ファイルの読み込みに使うD3DXCreateTextureFromFileEx関数は、テクスチャの縦横のサイズが2のn乗になるよう自動的に調整するため、テクスチャのサイズと元画像のサイズが異なる場合があります。調整を無効にすることもできますが、多くのグラフィックカードでゲームを動作させるには、テクスチャのサイズを2のn乗にしておく方が無難です（グラフィックカードによっては2のn乗以外のサイズのテクスチャを扱えない場合がありますから）。

## 2D画像の描画

テクスチャとして読み込んだ2D画像を描画するには、テクスチャをデバイスに設定した後に、矩形を描画します。つまり、テクスチャを適用したポリゴンを描画することによって、2D画像を表示するわけです。これは次のような手順で行います。

### テクスチャの設定

IDirect3DDevice9::SetTexture関数を使用して、ポリゴンに適用するテクスチャを選択します。また、IDirect3DDevice9::SetTextureStageState関数を用いて、ポリゴンの頂点色とテクスチャの色を合成する方法も指定しておく必要があります。

典型的な設定としては、頂点色とテクスチャ色を乗算する設定にします。例えば、頂点色を赤にして描画すると、赤みがかった画像を表示することができます。

### 頂点形式の表現

Direct3Dでポリゴンを描画するには、ポリゴンの頂点を表す構造体を定義する必要があります。構造体のメンバとしては、画面座標・色・テクスチャ座標などを用意します。

また、頂点形式を表す定数も用意します。Direct3Dでは、ユーザーがカスタマイズ可能な頂点形式のことを、FVF (Flexible Vertex Format) と呼んでいます。頂点形式を表す定

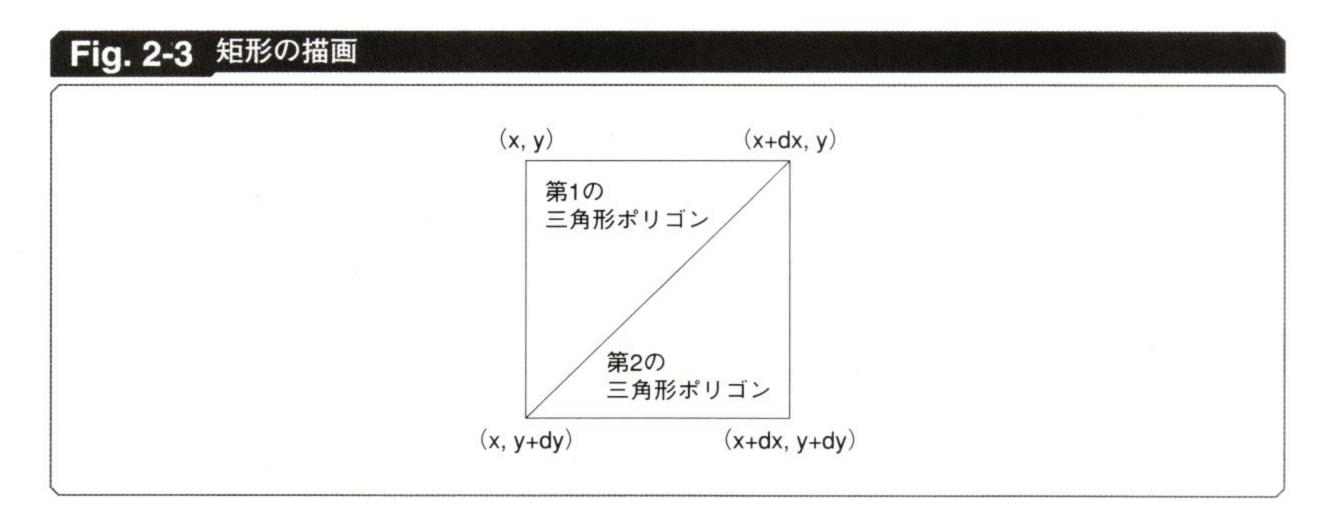
数は、FVFコードと呼ばれます。このFVFコードは、D3DFVFから始まるDirect3Dのマクロを用いて定義します。

なお、Direct3Dでは、FVFのかわりに頂点宣言（Vertex Declaration）という機能を使うこともできます。頂点宣言ではFVFよりも柔軟に頂点形式を表現することが可能です。

### 矩形の描画

2枚の三角形ポリゴンを描画することによって、矩形を描画します。最初に、前述の頂点形式を使って頂点を定義します。次に、IDirect3DDevice9::SetFVF関数でFVFコードを設定し、IDirect3DDevice9::DrawPrimitiveUP関数でポリゴンを描画します。

矩形を描画するときには、トライアングルストリップ（1辺を共有する三角形ポリゴンの連鎖）を使うとよいでしょう。矩形は2つの三角形の連鎖として描画できます（Fig. 2-3）。

**Fig. 2-3 矩形の描画**

## 矩形の描画

テクスチャを適用しないで矩形を描画すれば、2D画像ではなく、ただの矩形を描くことができます。ゲームの場合、ただの矩形はボスの耐久力ゲージなどを描画するときに便利です（P. 286）。

ただの矩形を描画するには、頂点色とテクスチャ色の合成方法を変更して、頂点色のみを使うように設定します。これはIDirect3DDevice9::SetTextureStageState関数を使います。次に、2D画像の描画と同様の処理を行って、矩形を描画します。

List 2-2は、矩形を描画するプログラムです。プログラムを少し拡張すれば、グラデーションを用いて矩形を描画することもできます。通常の描画では、頂点座標を設定する際に頂点色をすべてdiffuseにしていますが、これを頂点ごとに変えればよいのです。すると、頂点間は頂点色が線形補間された色になり、グラデーションを用いた矩形となります。

※

2D画像の読み込みと描画に関する機能は「Texture.h」と「Texture.cpp」にまとめました。詳しくは、付録CD-ROMの「LibGame¥Texture.h」と「LibGame¥Texture.cpp」をご参照ください。

**List 2-2 矩形の描画（Texture.cpp）**

```
void CTexture::DrawRect(
    LPDIRECT3DDEVICE9 d,
    float x, float y, float dx, float dy,
    D3DCOLOR diffuse
) {
    // 頂点色のみを使うように設定
    d->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_SELECTARG2);
    d->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
    d->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG2);
    d->SetTextureStageState(0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE);

    // 矩形の描画
    DrawRect(d, x, y, dx, dy, 0, 0, 0, 0, diffuse);
}
```

# フォントの読み込みと描画

ゲームではスコアやメッセージを表示する必要があります。ここでは、こういった文字情報の表示に便利なフォントのクラス（CFont）を作ってみました。このクラスでは、Fig. 2-4のような画像を使ってフォントを表示します。

GDIとDirectXを併用すれば、漢字や仮名を含んだ多種多様なフォントを表示することも可能です。GDI（Graphics Device Interface）というのは、Windowsアプリケーションにおける標準的なグラフィック描画機能です。

しかし、シューティングゲームの場合には、GDIを使うよりもFig. 2-4のようにフォントの画像を用意しておいた方が、簡単かつ高速に文字情報を表示できます。シューティングゲームでは必要な文字の種類が少ないので、このように画像としてフォントを用意することも十分に可能です。

CFontクラスを実際に使用したゲームプログラムの例は、Chapter 6で解説します（P. 185）。本書のサンプルでは、スコアやメッセージの描画にCFontクラスを使います。

**Fig. 2-4** フォントの画像

## フォントの初期化

まずはフォントの画像ファイルを読み込む必要があります。画像のロードには、前述のテクスチャクラス（CTexture）を使うのが簡単です（P. 32）。フォントの描画にも同じクラスの機能を使います。

次に、各キャラクタコードに対応するテクスチャ上の座標を計算して、配列などに保存します。フォントを初期化するときには、例えば次のような文字列を与えます。

```
~|0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ,.(){}<>!?+-x*/='¥"#$%&@[]
```

この文字列はFig. 2-4のフォント配置に対応しています。この文字列を用いて、各キャラクタに対応するテクスチャ上の座標を計算します。

## フォントの描画

フォントの描画については、フォントを普通に描画する関数に加えて、影つきで描画する関数を用意してみました。Fig. 2-5は影なしの描画と影つきの描画の例です。

**Fig. 2-5** フォントの描画

### 文字の影なし描画

1文字を描画します。テクスチャのなかで指定された文字のフォントに対応する部分だけを、指定された座標に描画します。

### 文字の影つき描画

1文字を影つきで描画します。文字を影なしで描画する関数を使用して、まず影を描画し、次に文字の本体を描画します。

影を表現するには、影と文字の位置を少しずらして描画します。影の位置や色は引数で指定できるようにするとよいでしょう。暗めの色を使うと、影らしい雰囲気が出ます。アルファブレンディングを用いて半透明で影を表示すると、さらに効果的です。

### 文字列の影なし描画

文字列を描画します。文字列から文字を1つずつ取り出し、画面に描画します。

### 文字列の影つき描画

文字列を影つきで描画します。文字列を影なしで描画する関数を使用して、文字の影と本体を描画します。

※

フォントの読み込みと描画に関する機能は「Font.h」と「Font.cpp」にまとめました。詳しくは、付録CD-ROMの「LibGame¥Font.h」と「LibGame¥Font.cpp」をご参照ください。

## 入力の読み取り

次に、キーボードやジョイスティックの入力を読み取る処理を作ります。入力の処理はインプットクラス（CInput）にまとめました。

キーボードやジョイスティックからの入力には、Win32 APIを使う方法と、DirectInputを使う方法とがあります。Win32 APIを使うことの利点は、処理が簡単なことです。一方、DirectInputを使うと、Win32 APIを使う場合よりも多くのスティックやボタンを制御することができます。本書ではDirectInputを使うことにしました。

インプットクラスを実際に使用したゲームプログラムの例は、Chapter 4で解説します（P. 105）。本書のサンプルでは、自機の移動やショットの発射などにこのクラスを使います。

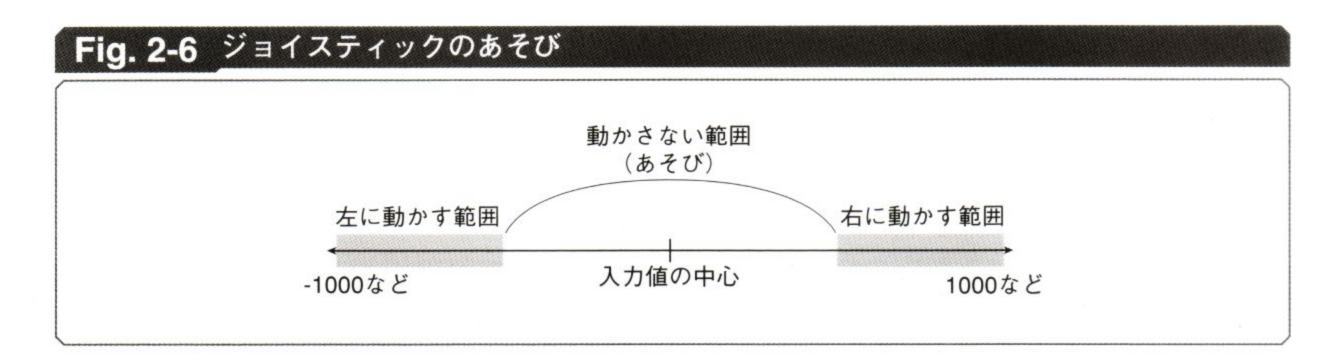
## ジョイスティックのあそび

DirectInputによるジョイスティック入力では、アナログスティックが想定されています。そのため、入力値は「0または1」ではなく、例えば「-1000から1000」のように幅のある値になります。

このとき、例えば入力値が負のときに自機を左を動かし、入力値が正のときに自機を右に動かすといったプログラムにすると、問題が生じます。おそらく、スティックを倒していないのに自機が動いてしまう、という結果になるでしょう。

一般にアナログスティックは、傾いていないときにも正確に中央の値を返すわけではなく、左右に少しずれた値を返します。スティックを倒していないからといって0を返すのではなく、負や正の値を返すことがほとんどです。

この問題を回避するためには、スティックの入力値がある程度中央に近いときには、自機を動かさないようにします（Fig. 2-6）。この動かさない範囲のことを「あそび」と呼びます。これは自動車のハンドルのあそびと同じことです。あそびを設けておかないと、スティックを倒していないのに自機が動いてしまう危険性があります。

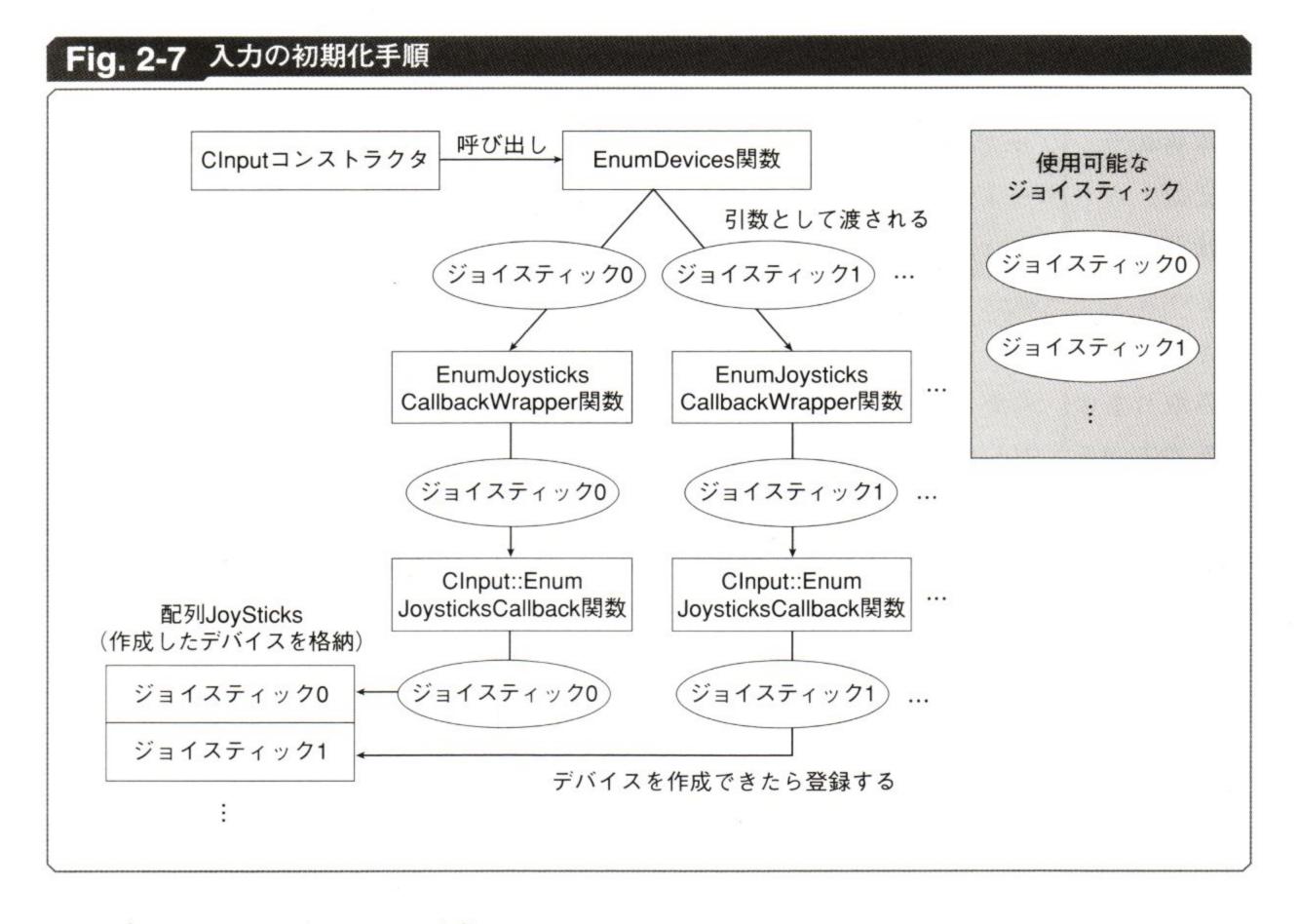
**Fig. 2-6** ジョイスティックのあそび

## 入力の初期化

最初にDirectInputの初期化を行います。DirectInputは提供する機能が単純かと思いきや、意外に初期化が大変です。ここでは大まかな初期化の流れを解説します。

### デバイスの初期化

最初にIDirectInput8インスタンスを作成します。次に、キーボードデバイスとゲームコントローラデバイスの取得および初期化を行います。

DirectInputでは、ジョイスティック・ジョイパッド・ハンドル・操縦桿のような装置を、ゲームコントローラと総称しています。本書ではゲームコントローラとして、主にジョイスティックとジョイパッドを想定しています。

Fig. 2-7 入力の初期化手順

## ジョイスティックの列挙

PCにどんなジョイスティックが接続されているのかを調べるためには、ジョイスティックの列挙という作業が必要です。ジョイスティックの列挙にはコールバック関数という仕組みを使います（Fig. 2-7）。

ジョイスティックの列挙を行うには、IDirectInput8::EnumDevices関数を呼び出します。このとき、コールバック関数を登録しておきます。

DirectInputはジョイスティックを検索して、使用可能なジョイスティックを1つ見つけるたびに、登録したコールバック関数を呼び出してくれます。コールバック関数は、発見したジョイスティックを扱うためのデバイスインスタンスを作成して、配列に登録します。配列に登録するのは、後でデバイスを利用しやすくするためです。

これでプログラムからジョイスティックが利用可能になります。

## スティックの範囲設定

1つジョイスティックを取得するたびに、そのジョイスティック上のボタンやスティックを列挙する作業を行います。このとき、スティックについては入力値の範囲を設定します。

## 入力の読み取り

本書のサンプルでは、キーボードとジョイスティックの入力を1つにまとめています。例えば、キーボードの「↑」キーを押すと、ジョイスティックの上方向を入力したのと同じ状態になります。入力をライブラリのレベルではまとめずに、ゲームアプリケーションのレベルでキーボードとジョイスティックからの入力を個別に処理する方法もあります。

キーボードとジョイスティックのボタンに関しては、各キーやボタンがオンかオフかを読み取ります。スティックに関しては、あそびを考慮して、入力範囲の両端付近のときだけその方向を入力したと判定することにしました。

List 2-3は、入力を読み取る処理です。キーボードとジョイスティックの入力状態を調べて、CInputStateというクラスに格納します。CInputStateについては、「Input.h」と「Input.cpp」をご覧ください。

※

入力の読み取りに関する機能は「Input.h」と「Input.cpp」にまとめました。詳しくは、付録CD-ROMの「LibGame¥Input.h」と「LibGame¥Input.cpp」をご参照ください。

**List 2-3　入力の読み取り（Input.cpp）**

```
// 状態読み取り用のマクロ
#define KEYDOWN(key) ((key_state[key]&0x80)!=0)
#define BTNDOWN(btn) ((joy_state.rgbButtons[btn]&0x80)!=0)

// 入力の読み取り
void CInput::UpdateState() {
    ClearState();

    // キーボード
    if (Keyboard) {
        char key_state[256];
        if (!FAILED(Keyboard->Acquire()) &&
            !FAILED(Keyboard->GetDeviceState(
                sizeof(key_state), key_state))
        ) {
            CInputState& s=State[0];
            s.Up|=KEYDOWN(DIK_UP);
            s.Down|=KEYDOWN(DIK_DOWN);
            s.Left|=KEYDOWN(DIK_LEFT);
            s.Right|=KEYDOWN(DIK_RIGHT);
            s.Button[0]|=KEYDOWN(DIK_Z);
            s.Button[0]|=KEYDOWN(DIK_SPACE);
```

```
            s.Button[0]|=KEYDOWN(DIK_RETURN);
            s.Button[1]|=KEYDOWN(DIK_X);
            s.Button[2]|=KEYDOWN(DIK_C);
            s.Button[3]|=KEYDOWN(DIK_V);
            s.Button[4]|=KEYDOWN(DIK_B);
            s.AnalogY+=(s.Up? -1 : (s.Down? 1 : 0));
            s.AnalogX+=(s.Left? -1 : (s.Right? 1 : 0));

            // ...(中略)...

        }
    }

    // ジョイスティック
    for (int i=0, n=JoySticks.size(); i<n; i++) {
        DIJOYSTATE2 joy_state;
        JoySticks[i]->Poll();
        if (!FAILED(JoySticks[i]->Acquire()) &&
            !FAILED(JoySticks[i]->GetDeviceState(
                sizeof(joy_state), &joy_state))
        ) {
            CInputState& s=State[i];
            s.Up|=(joy_state.lY<-JOYAXIS_MARGIN);
            s.Down|=(joy_state.lY>JOYAXIS_MARGIN);
            s.Left|=(joy_state.lX<-JOYAXIS_MARGIN);
            s.Right|=(joy_state.lX>JOYAXIS_MARGIN);
            s.Button[0]|=BTNDOWN(0);
            s.Button[1]|=BTNDOWN(1);
            s.Button[2]|=BTNDOWN(2);
            s.Button[3]|=BTNDOWN(3);

            // ...(中略)...

        }
    }
}
```

## 効果音の読み込みと再生

音があるのとないのとでは、ゲームの楽しさがまったく違います。シューティングゲームの場合には、ショットやビームの発射、爆発、アイテムの取得、ボムの使用などの際に効果音を再生します。Chapter 4では、ショットやビームの発射時に効果音を再生する方法を紹介します（P. 140）。

効果音の再生用にサウンドクラス（CSound）とサウンドプレイヤークラス（CSound Player）を作りました。前者は効果音データを管理するクラスで、後者は効果音を再生するクラスです。

効果音の再生にはDirectMusicを使いました。DirectMusicはWaveファイル（.wav）やMIDIファイル（.mid）の再生ができるAPIです。DirectMusicはDirectSoundの上位ライブラリという位置づけで、ファイルのロード機能など、DirectSoundにはない便利な機能があります。

DirectMusicを使って効果音を再生する手順は次のとおりです。

① COMの初期化
② DirectMusicの初期化
③ パフォーマンスの作成
④ オーディオパスの取得
⑤ DirectSound3Dバッファの取得
⑥ 効果音データをセグメントにロード
⑦ セグメントをパフォーマンスに設定
⑧ セグメントの再生

パフォーマンス、オーディオパス、セグメントなどはDirectMusicの用語です。詳細はDirectMusicのリファレンスをご覧ください。

なお、最近のDirectX UpdateではDirectMusicのサポートは終了しています。しかし、効果音を手軽にロードして再生できる便利なAPIなので、本書ではDirectMusicを使っています。DirectMusicのかわりにDirectSoundを使ってもかまいません。

サウンドクラスやサウンドプレイヤークラスを実際に使用したゲームプログラムの例は、Chapter 4で解説します。本書のサンプルでは、ショットの発射時や敵の爆発時などに、これらのクラスを用いて効果音を再生します。

### 効果音再生のための初期化

最初にCOM（Component Object Model Technologies）とDirectMusicの初期化を行います。COMというのは、Microsoftが提唱するソフトウェアコンポーネント技術の1つです。DirectMusicはCOMを利用して構築されています。

次に、パフォーマンスの作成、オーディオパスの取得、DirectSound3Dバッファの取得を行います。パフォーマンスは、音声の再生を管理するオブジェクトです。オーディオパスはDirectSound3Dバッファの取得時と、効果音の再生時に使用しています。DirectSound3Dバッファは、3Dサウンド（効果音を3D空間内で鳴らすようなエフェクト）のために使います。

### 効果音のロード

効果音データはファイルから読み込んで、セグメントにロードする必要があります。なお、DirectMusicでデータをロードする場合には、ファイル名をUnicode文字列で指定する必要があります。

### 効果音の再生

効果音を再生するには、セグメントをパフォーマンスにダウンロードしてから、セグメントを再生します。セグメントを再生する前には、必ず一度パフォーマンスにダウンロードしなければなりません。

※

効果音の読み込みと再生に関する機能は「Sound.h」と「Sound.cpp」にまとめました。詳しくは、付録CD-ROMの「LibGame¥Sound.h」「LibGame¥Sound.cpp」をご参照ください。

## BGMの読み込みと再生

効果音と同様に、BGMもゲームを盛り上げる大切な要素です。そこで、BGMを再生するためのクラスも作ります。

BGMの再生に効果音と同じサウンドクラスやサウンドプレイヤークラスを使うこともできます。しかし、これらのクラスはDirectMusicをベースにしているため、再生できる形式があまり多くありません。

そこで、DirectShowを使ってみることにしました。DirectShowはWaveやMIDIに加えて、

WMAやMP3といったさまざまな形式の音楽ファイルを再生することができます。また、動画ファイルを再生することも可能です。

DirectShowで音楽を再生する手順は以下のとおりです。

① COMの初期化
② グラフビルダの初期化
③ メディアコントロールの初期化
④ グラフの作成
⑤ グラフの実行

グラフ、グラフビルダ、メディアコントロールなどはDirectShowの用語です。詳細はDirectShowのリファレンスをご覧ください。

なお、最新のDirectX UpdateにはDirectShowは含まれておらず、Platform SDKの1要素となっています。Platform SDKは一般的なWindowsアプリケーションを作成するためのAPI群です。

音楽を再生する機能はメディアクラス（CMedia）にまとめました。本書のサンプルでは、BGMの再生にこのクラスを使います。BGMの再生についてはChapter 8で解説します（P. 271）。

## 音楽の再生

DirectShowで音楽を再生するには、最初にCOMを初期化し、次にグラフビルダを初期化します。続いて、グラフビルダのメンバ関数IGraphBuilder::RenderFile関数を使って音楽ファイルを読み込み、音楽を再生するためのグラフを作成します。

DirectShowでは、フィルタと呼ばれるオブジェクトを組み合わせてグラフを構築します。グラフの構成を変えることによって、さまざまなメディアを再生・録音することができます。IGraphBuilder::RenderFile関数を使うと、ファイルを再生するためのグラフを自動的に構築させることができます。

次に、メディアコントロールを初期化します。メディアコントロールを使うと、音楽の再生・停止・一時停止などを制御することができます。

その他、IMediaEventやIMediaSeekingといったオブジェクトの初期化も行います。これらは再生の終了を検出したり、再生位置を変更するために使います。詳細は「Media.h」と「Media.cpp」をご覧ください。

ここまでの準備ができたら、IMediaControl::Run関数を使って音楽を再生します。

※

音楽の読み込みと再生に関する機能は「Media.h」と「Media.cpp」にまとめました。詳しくは、付録CD-ROMの「LibGame¥Media.h」と「LibGame¥Media.cpp」をご参照ください。

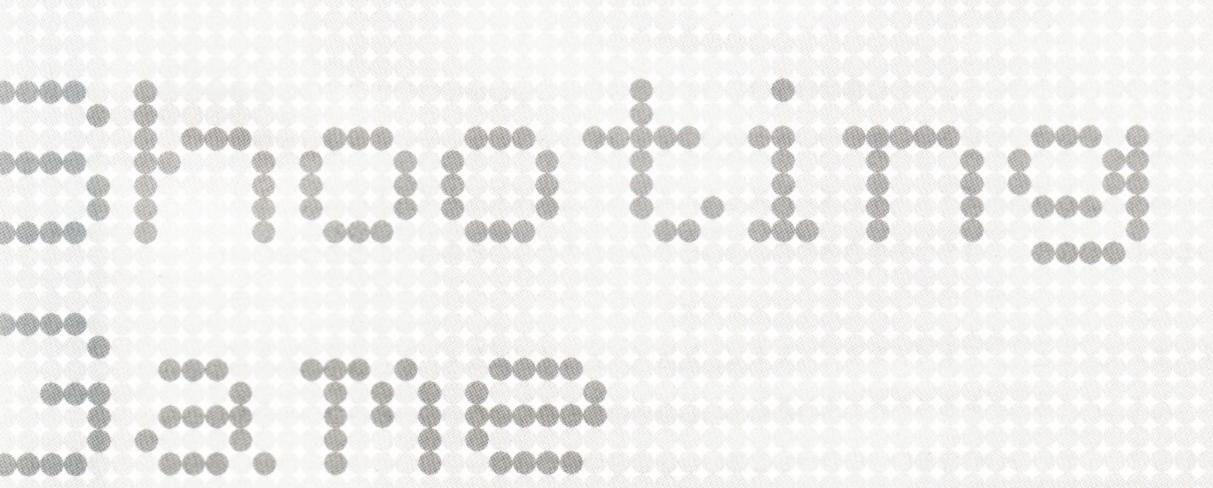
## >>Chapter 2のまとめ

本章ではシューティングゲームの制作に必要なDirectXの機能をまとめて、オリジナルのゲームライブラリを作りました。けっこうな大仕事になってしまいましたが、どんな処理をやっているのかをなんとなく眺めていただければ十分です。

このライブラリで提供しているような機能を自分でゼロから実現するとなると、ゲーム本体の制作以前にかなり時間をとられてしまいます。本書のライブラリを利用して、スムーズにゲーム制作にとりかかっていただければ幸いです。制作したライブラリの使用方法に関しては、Chapter 4以降で実際にシューティングゲームを作りながら紹介します。

# Chapter 03

# タスクシステム

タスクシステムはゲームプログラムで使う仕掛けの1つです。ジャンルを問わずいろいろな種類のゲームに適用できますが、特にシューティングゲームやアクションゲームに向いています。
本章では、タスクシステムの仕組みを解説し、タスクシステムを実装する方法を説明します。C言語の構造体と関数を使って実装する方法と、C++のクラスを使って実装する方法を両方紹介し、比較します。どちらの方法で実装するか悩んでいる方は、ぜひ参考にしてください。
なお、次章以降ではクラスを使ったタスクシステムを利用します。

# タスクシステムとは

タスクシステムの由来についてははっきりしませんが、ナムコのギャラクシアン（1979年）で開発されたという説があります。その後、他社によるリバースエンジニアリングなどを通じて、ゲームプログラマーの間に広がっていったという噂です。

タスクシステムは複数のタスク（仕事）を並行して処理するための仕掛けです。タスクシステムを使ったゲームプログラムでは、ゲームに登場する各種の要素をタスクで表現します。例えばシューティングゲームならば、自機・敵・弾といったキャラクターをそれぞれタスクとして実装します。

タスクはスレッドのようなものです。ゲームでは多くのキャラクターが並行して動きますが、タスクシステムを使うことによって、このような処理を自然な形で表現することができます。また、タスクシステムが提供するタスクの生成や削除といった機能は、「弾を撃つ処理」や「破壊した敵を消す処理」などの実現に役立ちます。

なお、本書では便宜上タスクシステムという名称を使いますが、これは情報科学の正式な用語ではないようです。ゲーム業界用語といったところでしょうか。ゲームによっては同じような仕組みのことを別の名前で呼んでいる場合もあります。

## タスクシステムは本当に必要か

実は、タスクシステムはゲームプログラムに不可欠な仕組みではありません。「ゲームを作るには、まずタスクシステムを作らなくてはならない」と本で読んだり人に教わったりすることがあるかもしれませんが、あせってタスクシステムについて勉強しなくても大丈夫です。

ゲームプログラミングを楽しむうえで大事なのは、自分の好きな流儀でプログラムを書くことと、ゲームを完成させることだと思います。そして、自分が作りたいゲームに合った実装方法を追究することも重要です。もしもタスクシステムに馴染みがなければ、そんなものの存在はきれいに忘れて、自分が書きやすい方法でプログラムを書くことをお勧めします。

「ゲームを作るには、なんとしてもタスクシステムを作らなければならない」と考える必要はありません。たしかにタスクシステムは便利な仕組みですが、タスクシステムの制作でつまずいてしまって、ゲームが完成しなければ本末転倒です。

そもそもゲームプログラムというものは、

・プレイヤーの入力を受け取る
・キャラクターを動かす
・当たり判定処理やスコアの加算を行う
・キャラクターやスコアを表示する

くらいのごく簡単な処理を繰り返すだけなのです。素朴なプログラムからスタートして、複雑な仕掛けが本当に必要になったら、そのときに初めて作ればよいのです。

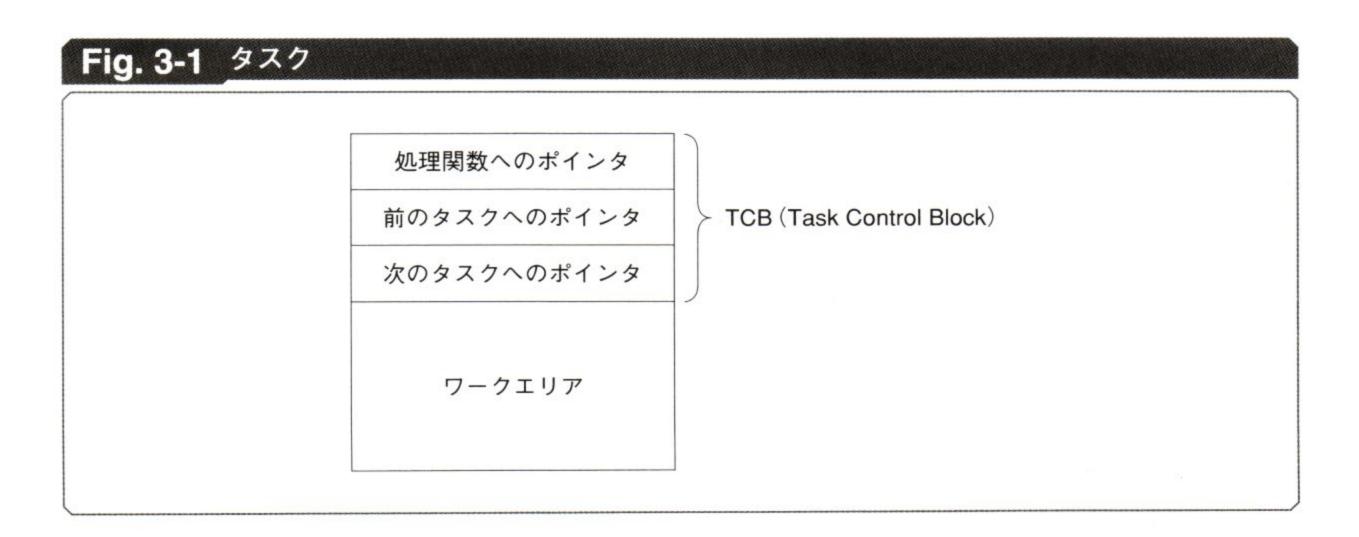
次章以降で解説するゲームは、タスクシステムを利用しています。スムーズな理解のために、まずは本章にざっと目を通してみてください。そして後になって、いざタスクシステムを作成したくなったときに、本章を詳しく読んでいただくのがよいと思います。

## タスク＝処理関数ポインタ＋ワークエリア

タスクは、関数ポインタ（処理関数へのポインタ）とワークエリア（作業用のデータ領域）を組み合わせたものです（Fig. 3-1）。それに加えて、複数のタスクを連結リストとして管理するために、前のタスクと次のタスクへのポインタも持ちます。

処理関数というのはそのタスクの処理を行うための関数です。必要に応じて処理関数へのポインタを書き換えることによって、タスクが行う処理の内容を変えることができます。

処理関数へのポインタと前後のタスクへのポインタとを合わせた領域は、TCB（Task Control Block）と呼ばれることがあります。これはタスクの制御に必要な情報を保持する領域です。タスクシステムによっては、この他にもタスクのIDや優先度といった情報をTCBに格納することがあります。

Fig. 3-1 タスク

## タスクシステムを実装する言語

タスクシステムの実装方法は言語やライブラリによって変わります。本章ではまず、C言語の構造体を使って実装する方法を解説します。これは簡単な実装方法の紹介と、タスクシステムのもともとの姿について説明するためです。

続いて、C++のクラスを使った実装方法を説明します。C++ではC言語に比べて名前空間をすっきりと整理でき、プログラムが簡潔になるという利点があります。C++に慣れているならば、こちらの方がお勧めの実装方法です。しかし、C++よりもC言語が手に馴染んでいる方は、まずは構造体を使った実装を試していただくのもよいでしょう。

C言語を使ったタスクシステムのプログラムは、付録CD-ROMの「TaskSystem」フォルダに収録しています。一方、C++を使ったプログラムは、「LibGame」フォルダに収録しています。次章以降では、C++を使ったタスクシステムを利用して、シューティングゲームを制作します。

## タスクの表現

C言語を使う場合には、タスクを構造体で実装するとよいでしょう。構造体には以下のような要素を用意します。

### 処理関数へのポインタ

処理関数の引数や戻り値の型はある程度自由に決めることができます。例えば、引数はタスク構造体へのポインタ、戻り値はなし、といった具合です。

### 前後のタスクへのポインタ

タスクは連結リストにして使うので、前後のタスクを指すポインタが必要です。

### ワークエリア

ワークエリアはタスクに関する情報を保存するために十分なサイズにします。例えば、100バイトや256バイトといった具合です。複雑な処理を行うには大きなワークエリアが必要ですが、タスク数が多い場合にはワークエリアを小さくする工夫も必要です。

※

List 3-1は、C言語の構造体で実装したタスクです。

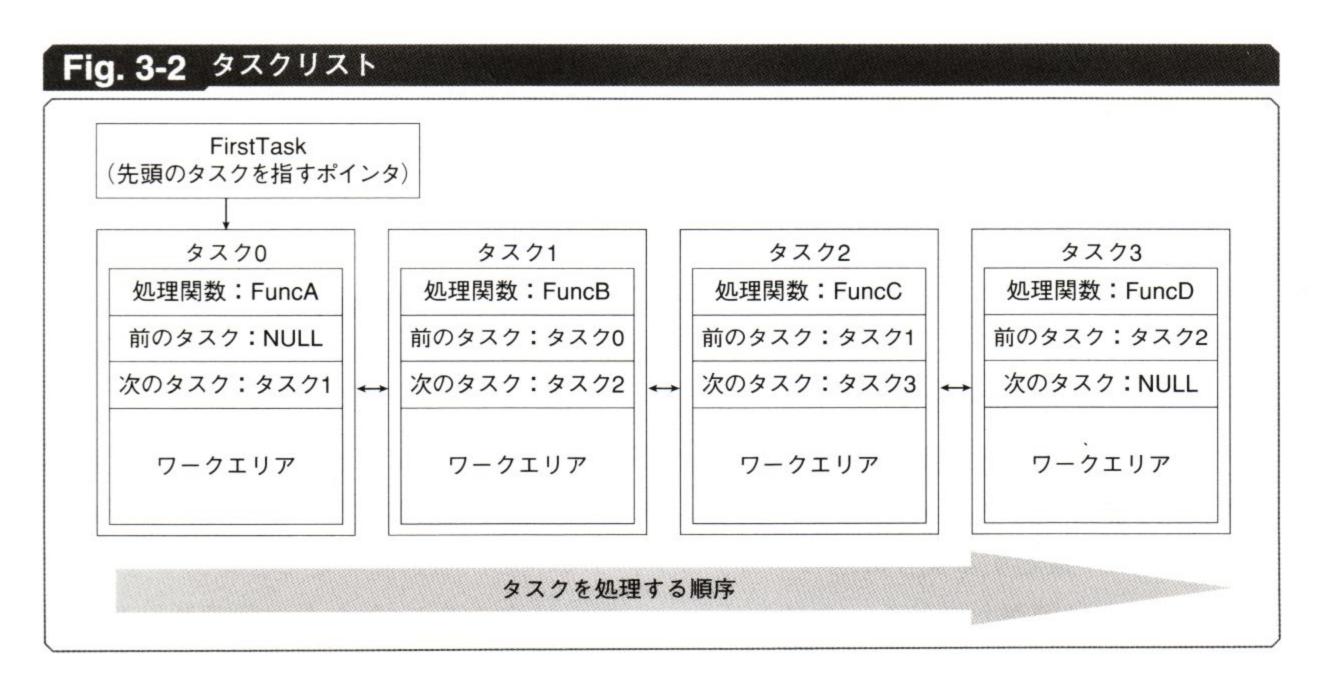
**List 3-1 タスクの構造体宣言 (Task1.cpp)**

```
// タスクの構造体
struct TASK {

    // 処理関数へのポインタ
    void (*Func)(TASK* task);

    // 前後のタスクへのポインタ
    TASK* Prev;
    TASK* Next;

    // ワークエリア
    // ここでは256バイトとした
    char Work[256];
};
```

## タスクシステム＝タスク＋連結リスト

タスクは連結して使います。各タスクが持っている前後のタスクへのポインタを使って、連結リストを作成することができます（Fig. 3-2）。ここではタスクの連結リストを「タスクリスト」と呼ぶことにします。

タスクシステムは、タスクリストの先頭から末尾に向かって、タスクの処理関数を順番に呼び出していきます。Fig. 3-2の場合には、最初にタスク0のFuncA関数を呼び出し、次

**Fig. 3-2 タスクリスト**

にタスク1のFuncB関数、続いてタスク2のFuncC関数、最後にタスク3のFuncD関数を呼び出します。

タスク3の次のタスクはないので、ここで1周期ぶんの処理を終えます。そして次の周期には、再び先頭のタスク0から同様に処理を行います。この手順を短い間隔（1/60秒など）で繰り返すことによって、ゲームを進行させます。

## 全タスクを実行する

タスクリストの全タスクを実行するには、タスクリスト上のタスクを1個ずつ順に取り出して、各タスクに登録された処理関数を呼び出します。それにはまず、タスクリストの先頭にあるタスクへのポインタを保持しておく必要があります。

そして、forループなどを使い、先頭のタスクから始めて、タスクがあるかぎり処理関数の呼び出しを続けます。こうして、全タスクの処理関数が呼び出されます。

このプログラムはList 3-2のようになります。なお、現在処理中のタスクへのポインタ（task）をグローバル変数にしておくと、Funcの引数にtaskを与える必要がなくなります。好みに応じてグローバル変数を使ってもよいでしょう。

**List 3-2　全タスクの実行（Task1.cpp）**

```
// タスクリストの先頭のタスクへのポインタ
TASK* FirstTask;

// タスクの実行
void RunTask() {
    for (TASK *task=FirstTask, *next;
        next=task->Next, task; task=next)
        (*task->Func)(task);
}
```

## タスクの追加・削除を効率化する

前節では「タスクリストの先頭を指すポインタ」を使ってリストを保持しましたが、実はもう少し処理がスマートになる方法があります。それは、タスクリストの先頭に「ダミータスク」を置く方法です（Fig. 3-3）。

ダミータスクは先頭および末尾のタスクとポインタで結合して、全体としてリストが循環するようにします。他のタスクとは違い、ダミータスクには実行すべき処理関数がありません。また、後述するように他のタスクはタスクリストから削除することがありますが、ダミータスクは削除しません。タスクリストのなかにタスクが1個もない状態でもダミー

タスクは残るので、ポインタがNULLになることがなくなります (Fig. 3-4)。

こうすると、リストの要素を追加または削除する際に、ポインタがNULLでないかどうかをチェックする必要がなくなります。つまり、条件分岐を1つ減らすことができます。これがダミータスクを使うことの利点です。ハードウェアやプログラムの性質によりますが、一般に条件分岐は実行速度を低下させる危険性をはらんでいます。タスクシステムでは、タスクの生成や削除のたびにリストの操作を行うので、NULLチェックをなくすことはシステム全体の性能向上につながります。

ダミータスクはタスク1個ぶんのメモリを消費しますが、数百ものタスクがタスクリストに登録されることを考えれば、1個くらいの無駄は無視できる範囲です。実際に消費するメモリ領域も、タスクのサイズによりますが、数十バイトから数百バイトといったケースが多いでしょう。シューティングゲームのような場合は、プログラム全体でタスク数個ぶんのメモリ消費を抑えることよりも、実行速度を重視した方が有利なので、以後のプログラム例ではダミータスクを使うことにしました。

ダミータスクを使った場合、タスクを実行するプログラムはList 3-3のようになります。実行に関してはダミータスクを使わない場合 (List 3-2) とあまり違いませんが、後述するタスクの生成や削除については、こちらの方がスマートなプログラムになります。

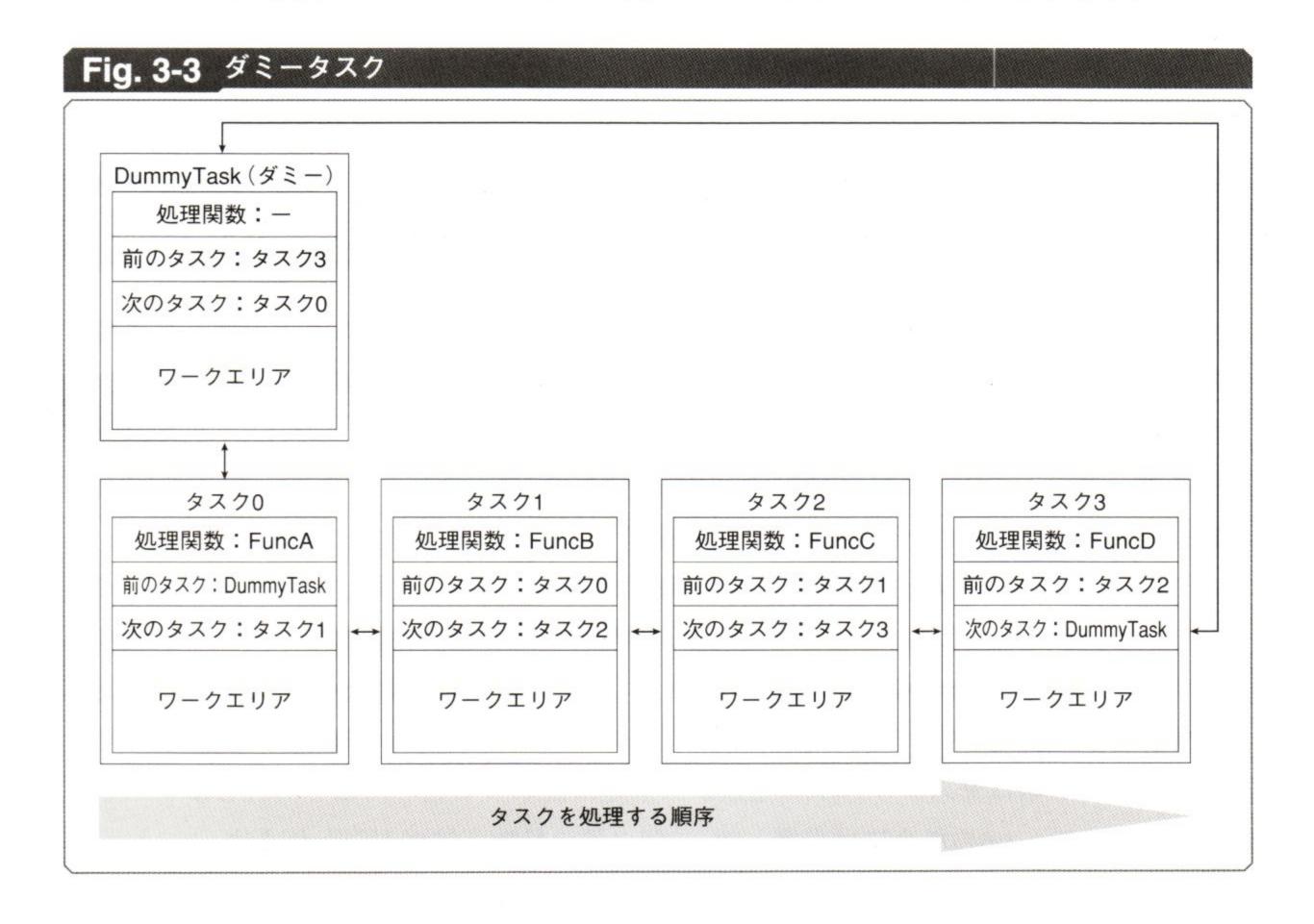
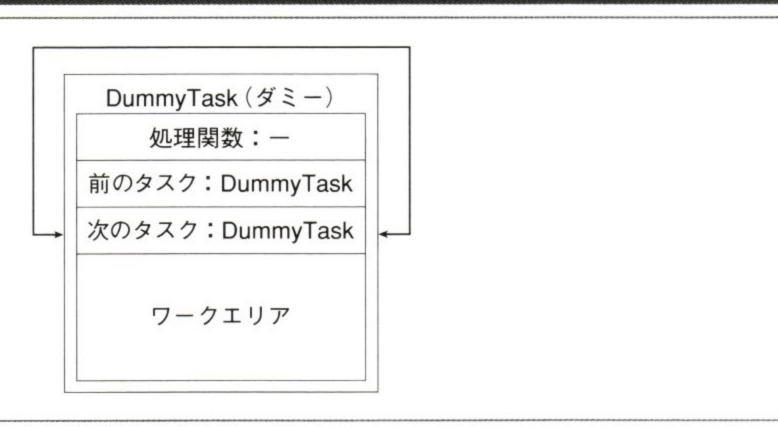



**Fig. 3-3** ダミータスク

**Fig. 3-4 タスクが1個もなくダミータスクのみの状態**

**List 3-3 ダミータスクを使う（Task1.cpp）**

```
// 先頭のタスク（ダミー）
TASK* DummyTask;

// タスクの実行
void RunTask() {
    for (TASK *task=DummyTask->Next, *next;
        next=task->Next, task!=DummyTask; task=next)
        (*task->Func)(task);
}
```

## タスクを使ってキャラクターを動かす

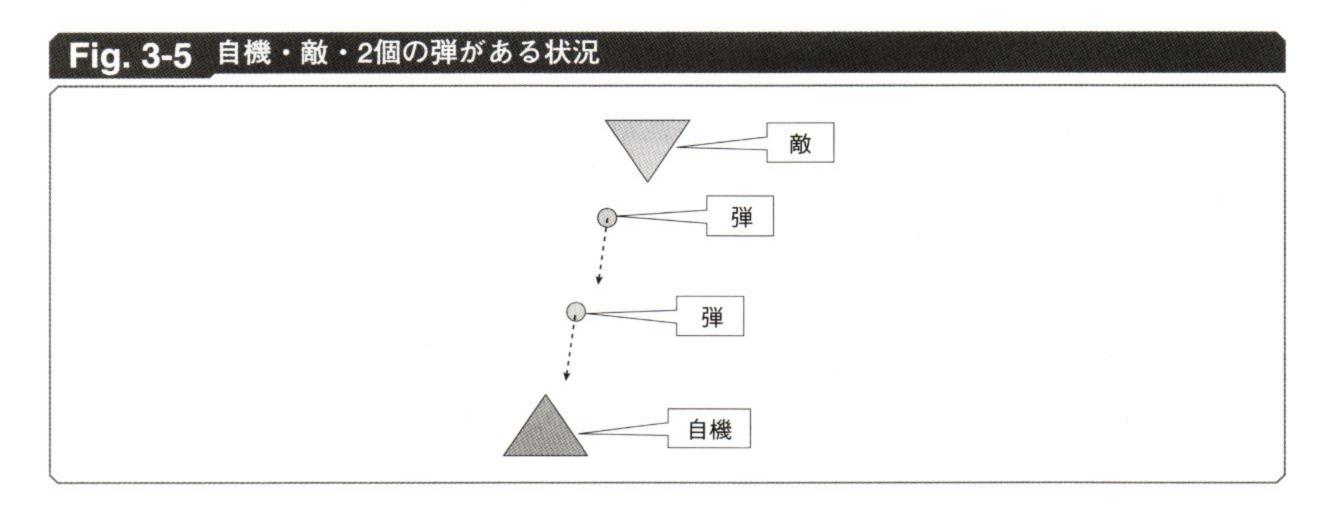
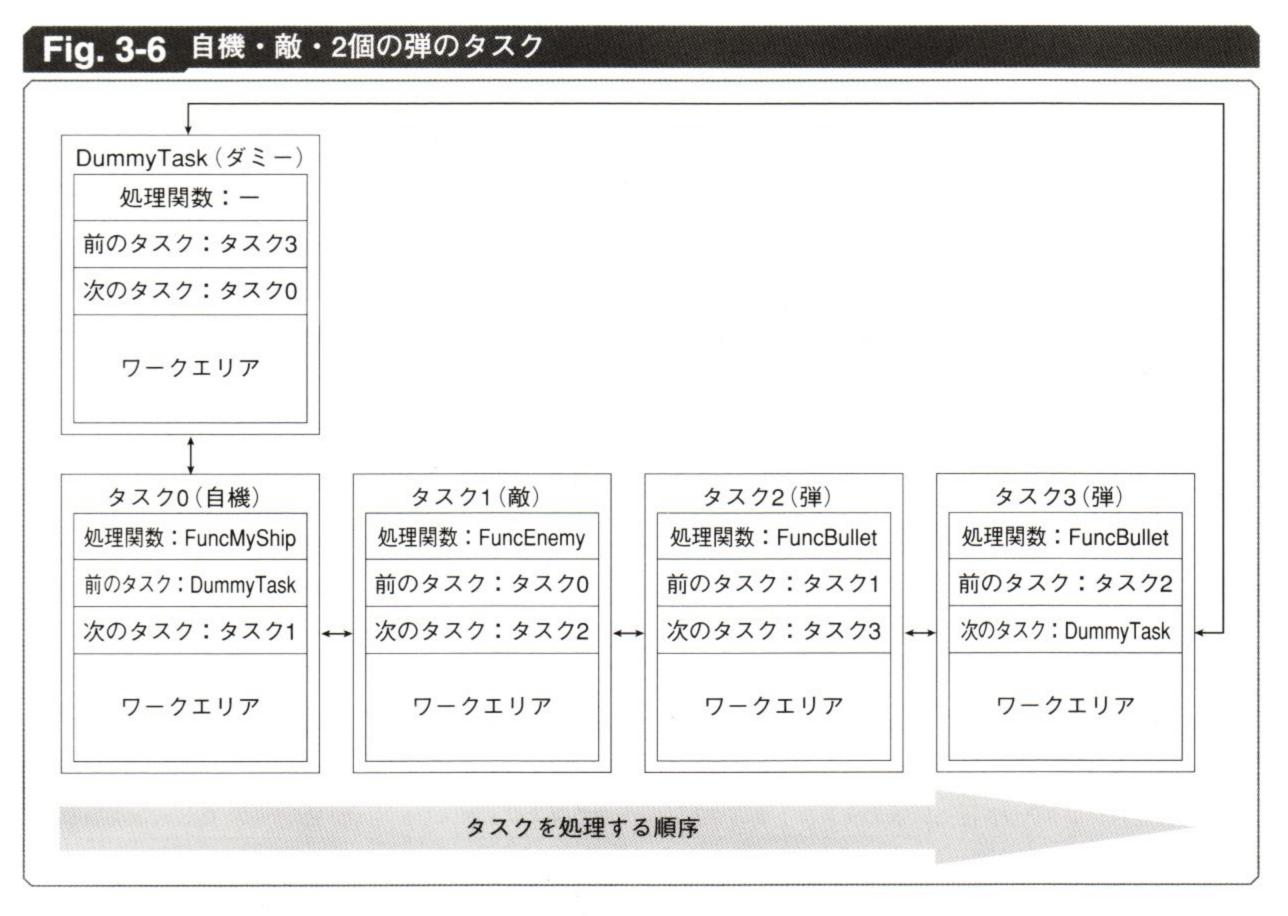
タスクシステムにおけるタスクの実行は、ゲームにおけるキャラクターの動きと密接に結びついています。ここではFig.3-5のように、自機と敵、そして2個の弾がある状況を考えてみましょう。

タスクシステムを使う場合、自機・敵・2個の弾を計4個のタスクで表現します(Fig. 3-6)。4個のタスクはタスクリストに格納されています。

ここでは、自機の移動と描画を行う処理関数を「FuncMyShip」とし、敵や弾についても同様に「FuncEnemy」と「FuncBullet」という処理関数を用意することにします。自機のタスク（タスク0）にはFuncMyShip、敵のタスク（タスク1）にはFuncEnemyをそれぞれ登録します。2個の弾のタスク（タスク2とタスク3）には、どちらもFuncBulletを登録します。

タスクシステムはタスクリストの先頭から末尾に向かって処理関数を実行します。その結果として、Fig. 3-6の場合には自機・敵・2個の弾の移動と描画が行われます。

**Fig. 3-5 自機・敵・2個の弾がある状況**

**Fig. 3-6 自機・敵・2個の弾のタスク**

## ワークエリアにキャラクターの情報を格納する

各タスクが持つワークエリアは、処理関数が自由に使うことのできるメモリ領域です。ワークエリアの使い方はタスクによって異なります。

例えば、シューティングゲームのキャラクターには、座標（例えばXとY）や速度（例えばVXとVY）といった情報が必要です。こういった情報はワークエリアに置きます。

また、自機にはパワーアップに関する情報が必要だったり、敵には耐久力の情報が必要だったりすることもあります。ワークエリアの使い方はタスクごとに違ってもかまわないので、タスクの種類に応じてワークエリアのなかに格納する情報を変えます。

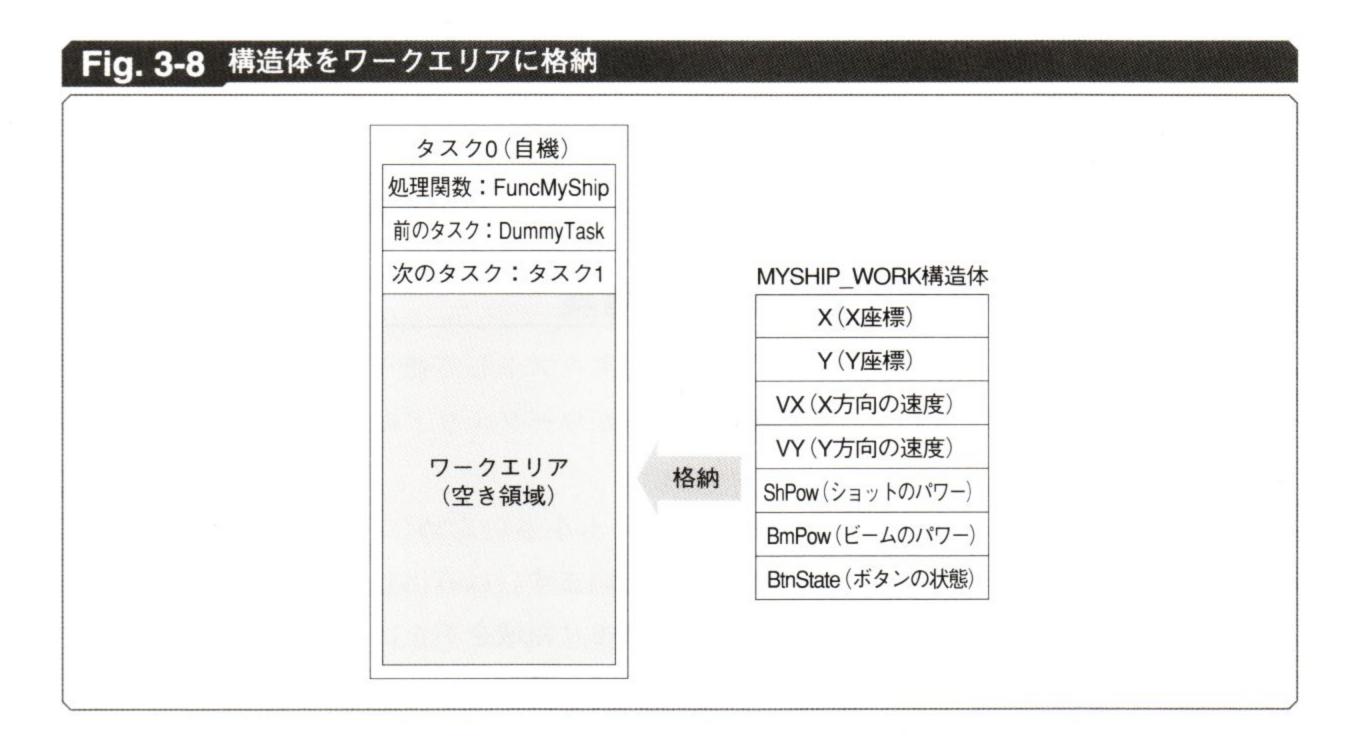
Fig. 3-7はワークエリアの使用例です。座標と速度はすべてのタスクに共通ですが、この他に自機のタスクにはショットやビームのパワー、敵のタスクには耐久力やスコアといった情報があります。

**Fig. 3-7** ワークエリアの使用例

タスクの種類によってワークエリアの内容を変えるには、構造体とキャストを使います。必要な情報を構造体に記録し、ワークエリアに格納します（Fig. 3-8）。例えば自機の場合には、座標や速度、ショットやビームの強さ、ボタンの状態といった情報を記録します。

自機の処理関数からワークエリアを参照するときには、汎用ワークエリアのアドレス（例えばchar [] 型）を、自機ワークエリア構造体へのポインタ（例えばMYSHIP_WORK*型）に変換します。ポインタを変換してしまえば、あとは自機ワークエリア構造体のメンバ名を使って、ワークエリアにアクセスすることができます。敵タスクや弾タスクの場合にも同様に、それぞれ専用の構造体を宣言しておき、ワークエリアのアドレスを構造体へのポインタにキャストして使います。

List 3-4は、自機用にワークエリアを使う例です。

**Fig. 3-8 構造体をワークエリアに格納**

**List 3-4 ワークエリアを使う（Task1.cpp）**

```
// 自機のワークエリア構造体
struct MYSHIP_WORK {

    // 座標と速度
    float X, Y, VX, VY;

    // ショットとビームのパワー
    int ShPow, BmPow;

```

```
    // ボタンの状態
    bool BtnState;
};

// 自機の処理関数
void FuncMyShip(TASK* task) {

    // 汎用のワークエリアへのポインタを、
    // 自機ワークエリア構造体へのポインタにキャストする
    MYSHIP_WORK* work=(MYSHIP_WORK*)task->Work;

    // 以下では自機ワークエリア構造体のメンバ名を使って、
    // 自機に関する処理を行う
    work->X+=work->VX;
    work->Y+=work->VY;

    // ...(中略)...

}
```

## ワークエリアのキャストに伴う危険

ワークエリアをタスクの種類に応じた構造体にキャストして使うことには、実は危険が伴います。問題が発生するのは、構造体のサイズがワークエリアのサイズを上回ってしまったときです(Fig. 3-9)。

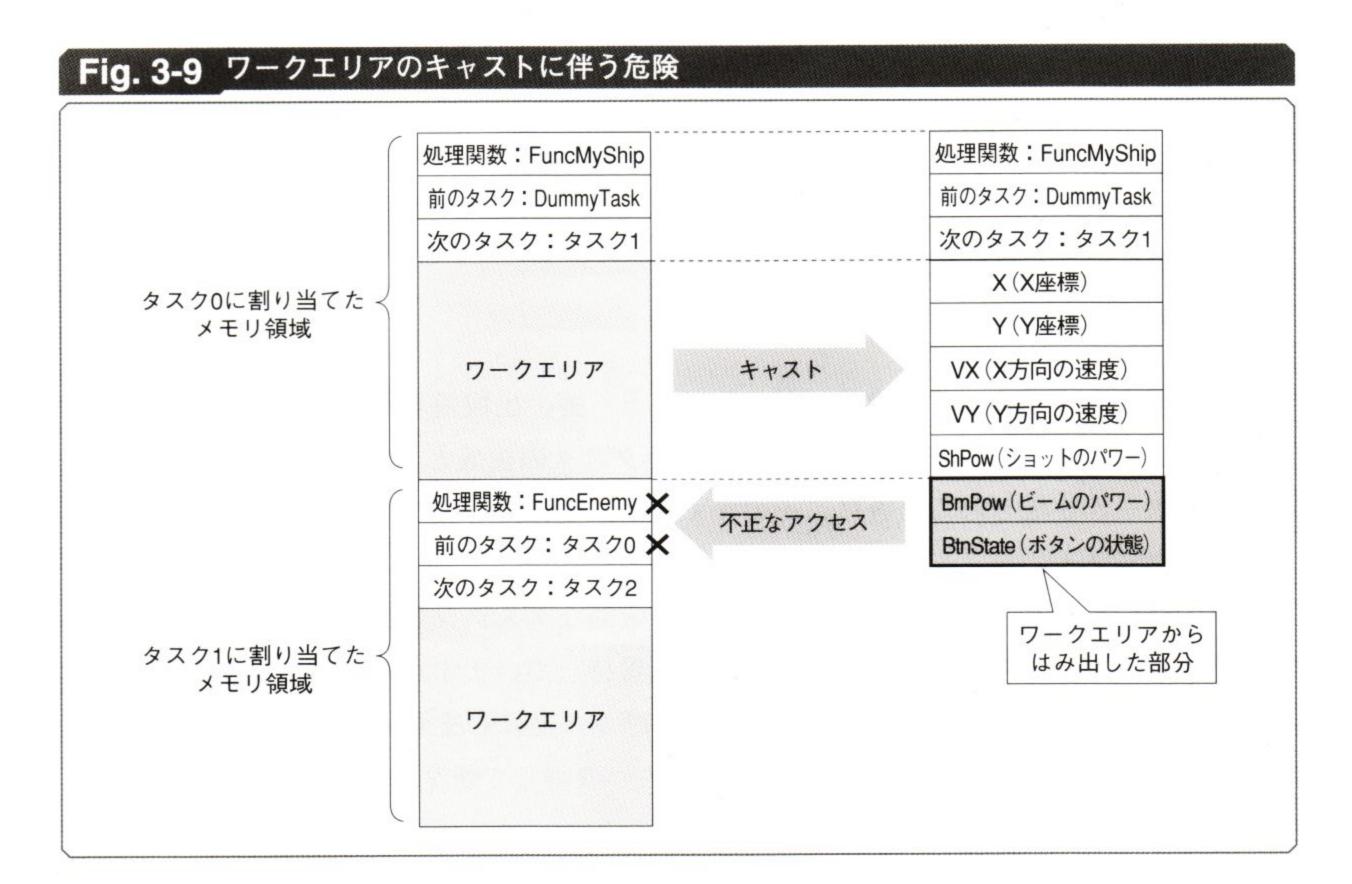
Fig. 3-9ではワークエリアが必要なサイズよりも小さいために、構造体の末尾にあるBmPowとBtnStateがワークエリアからはみ出しています。はみ出したメンバをうっかり読み書きすると、多くの場合は隣接するタスクのメモリ領域を不正に読み書きしてしまう結果になります。これは原因の特定が難しい深刻なバグです。

このバグを見つけやすくする方法の1つは、構造体のサイズがワークエリアのサイズ以下かどうかをチェックすることです。チェックにはassert関数などが使えます。大きすぎるワークエリアがあったらテスト時にすぐに検出できるようにしておいて、ゲーム公開前に問題を解決しておくのです。assert関数はプログラムのリリース時には削除されるので、チェックに伴う速度低下の心配はありません。

なお、メモリ上のデータを整列させる都合上、コンパイラは構造体のサイズを自動的に調整することがあります。そのため、構造体のサイズを手計算で求めてチェックするのは危険です。sizeofやassertなどの関数を使って、実行時にチェックすることをお勧めします。

List 3-5は、構造体のサイズをチェックする例です。デバッグモードで実行していると

きに構造体のサイズがワークエリアのサイズを超えると、メッセージを表示してプログラムの実行が停止します。

**Fig. 3-9 ワークエリアのキャストに伴う危険**

**List 3-5 構造体のサイズをチェックする（Task1.cpp）**

```
#include <assert.h>

// ワークエリアのサイズ
#define WORK_SIZE 256

// 自機の処理関数
void FuncMyShip(TASK* task) {

    // 構造体のサイズがワークエリアのサイズ以下かどうかを、
    // assert関数を使ってチェックする
    assert(sizeof(MYSHIP_WORK)<=WORK_SIZE);

    // 汎用のワークエリアへのポインタを、
    // 自機ワークエリア構造体へのポインタにキャストする
    MYSHIP_WORK* work=(MYSHIP_WORK*)task->Work;

    // ...（中略）...

}
```

# タスクの生成

ゲームにはキャラクターを生成する機会が多くあります。例えば、

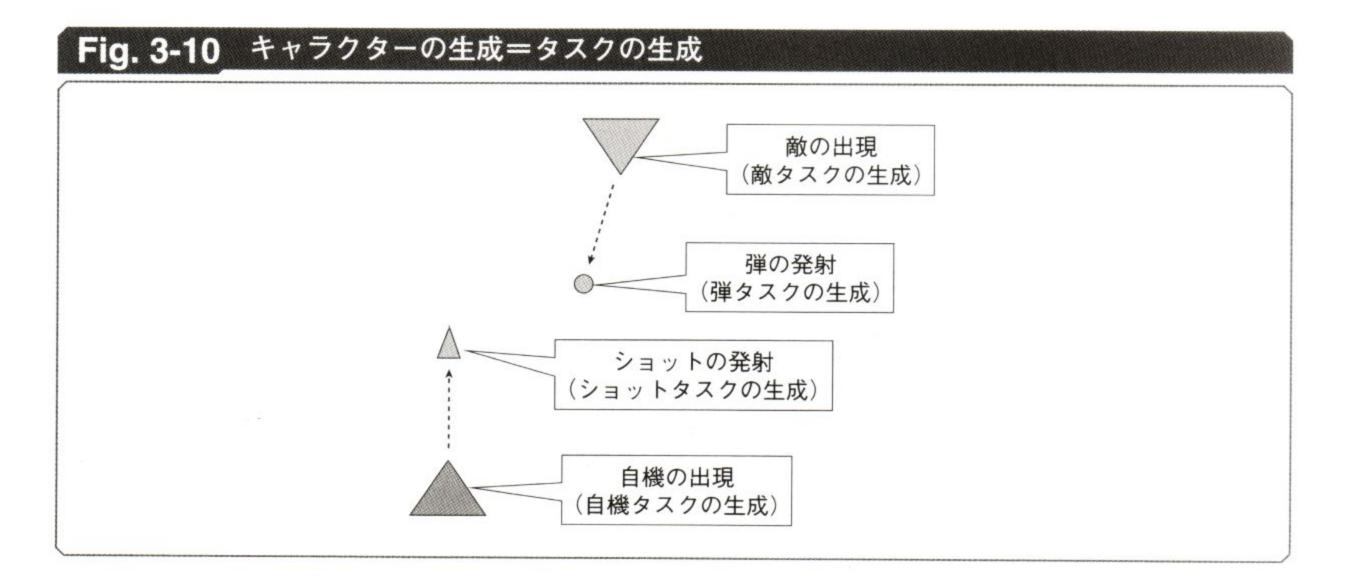
・自機を出す
・敵を出す
・自機がショットを撃つ
・敵が弾を撃つ

といった機会には、それぞれ自機・敵・ショット・弾の生成処理が行われます。タスクシステムでは、こういったキャラクターの生成をタスクの生成として扱います(Fig. 3-10)。キャラクターが生成されるときに、そのキャラクターに対応したタスクを生成してアクティブタスクリストに加えます。

Fig. 3-11〜13はタスク生成の仕組みです。一般的なタスクシステムでは、使用中のタスクのリストの他に、未使用のタスクのリストを保持しています。ここでは前者をアクティブタスクリスト、後者をフリータスクリストと呼ぶことにします(Fig. 3-11)。新しくタスクを生成するには、フリータスクをリストから1つ取得して新タスクに割り当てます。

アクティブタスクリストは双方向リストです。一方、フリータスクリストは双方向にする必要がないので、単方向としています。また、Fig. 3-11に「－」で示したメンバは使用しません。

さて、Fig. 3-11ではアクティブタスクリストに自機タスクと敵タスクがあります。フリータスクリストには3個のフリータスク(未使用タスク)があります。ここで新たにタスク

**Fig. 3-10** キャラクターの生成＝タスクの生成

を生成するには、フリータスクリストからタスクを1個取り出して、アクティブタスクリストに加えます。具体的には、フリータスクリストの先頭（ダミーの直後）にあるタスクを取り出して、アクティブタスクリストの末尾に移動します（Fig. 3-12）。

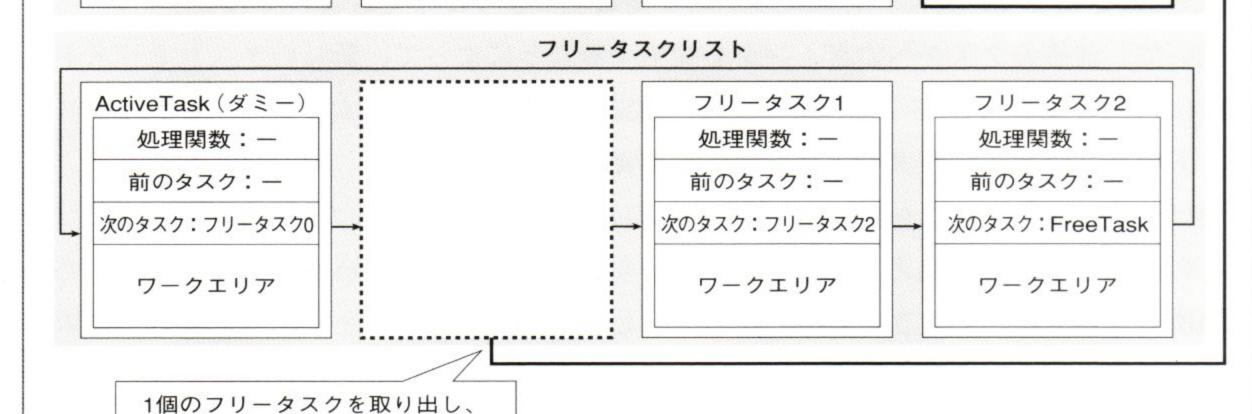
なお、ゲームに必要なタスク数をあらかじめ計算し、フリータスクは十分な数を用意しておきます。例えば、同時に出現する弾の最大数を1000個とすると、弾用にフリータスクが1000個必要です。その他、自機やショットなどに必要な数も計算しておきます。サンプルでは、フリータスクが不足すると、タスクを生成しないままゲームを進行させています。assert関数を使って、フリータスクの不足を検知するようにしてもよいでしょう。

続いて、前後のタスクへのポインタを変更することによって、新しいタスクをリストの末尾に結合します（Fig. 3-13）。処理関数の設定もあわせて行います。また、フリータスクリストのダミータスクのポインタも更新が必要です。

要はリストからタスクを1個取り出して別のリストにつなぎ直すだけの処理なのですが、多くのポインタを書き換えるのでバグが入りやすいところです。Fig. 3-13では変更が必要な箇所を太枠で示しておきました。なお、新しいタスクはアクティブタスクリストの末尾以外に挿入することもできます。例えばリストの先頭に入れたり、現在処理中のタスクの前後に入れたりといったことも可能です。

List 3-6は、1個のタスクを生成するプログラムです。

**Fig. 3-11　タスクを生成する前の状態**

## Fig. 3-12 生成するタスクの移動

## Fig. 3-13 タスク生成が完了した状態

**List 3-6 タスクを生成する（Task1.cpp）**

```
// 処理関数へのポインタ型
// 処理関数へのポインタは少し複雑な形なので、
// このようにtypedefしておくと便利
typedef void (*FUNC)(TASK* task);

// タスクリストの先頭要素（ダミー）
TASK* ActiveTask;
TASK* FreeTask;

// タスクの生成
TASK* CreateTask(FUNC func) {

    // フリータスクリストが空ならば生成を中止する
    if (FreeTask->Next==FreeTask) return NULL;

    // フリータスクを1個取り出す
    TASK* task=FreeTask->Next;
    FreeTask->Next=task->Next;

    // 処理関数と前後タスクへのポインタを設定する
    task->Func=func;
    task->Prev=ActiveTask->Prev;
    task->Next=ActiveTask;

    // 新しいタスクの前後のタスクに関して、
    // 前後タスクへのポインタを変更
    task->Prev->Next=task;
    task->Next->Prev=task;

    // 生成したタスクを返す
    return task;
}
```

## タスクリストの初期化

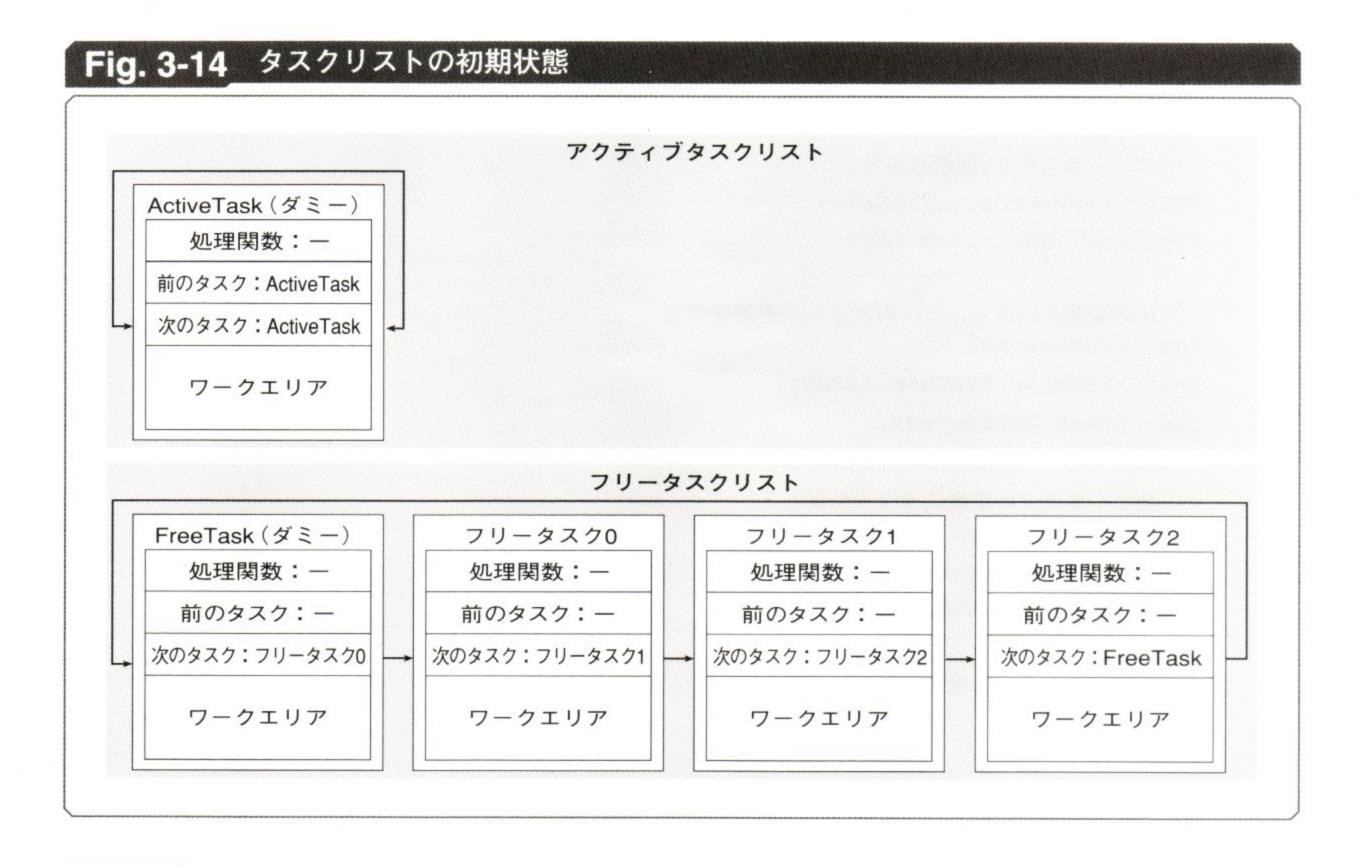
プログラムの開始時には、フリータスクリストとアクティブタスクリストを初期化しておく必要があります。Fig. 3-14は両タスクリストの初期状態です。アクティブタスクリストにはダミー以外のタスクがなく、フリータスクリストにはすべてのタスクがフリータスクとして登録されています。

このようにタスクリストを初期化するには、まずタスク用のメモリを確保します。2個のダミータスク（ActiveTask、FreeTask）のためのメモリもいっしょに確保するとよいでしょう。

次にアクティブタスクリストを初期化します。ActiveTask（ダミー）では、前後タスクへのポインタを自分自身に設定します。

続いてフリータスクリストを初期化します。各フリータスクが次のフリータスクを指すように、ポインタを設定します。Fig. 3-14に「－」で示したメンバは使用しないので、初期化は必要ありません。

List 3-7は、タスクリストを初期化するプログラムです。

**Fig. 3-14 タスクリストの初期状態**

**List 3-7 タスクリストの初期化（Task1.cpp）**

```
// タスク数
#define NUM_TASKS 1024

// タスクリストの初期化
void InitTaskList() {

    // タスク用メモリの確保
```

```
    TASK* task=new TASK[NUM_TASKS+2];

    // アクティブタスクリストの初期化
    ActiveTask=&task[0];
    ActiveTask->Prev=ActiveTask->Next=ActiveTask;

    // フリータスクリストの初期化
    FreeTask=&task[1];
    for (int i=1; i<NUM_TASKS+1; i++)
        task[i].Next=&task[i+1];
    task[NUM_TASKS+1].Next=FreeTask;
}
```

## タスクの削除

キャラクターの生成とは逆に、キャラクターを削除する機会もゲームにはたくさんあります。例えば次のような場合です。

- 破壊された自機を消す
- 自機に接触した弾を消す
- 破壊された敵を消す
- 敵に接触したショットを消す
- 画面外に出たショットを消す
- 画面外に出た弾を消す
- 画面外に出た敵を消す

これらはそれぞれ、自機・敵・ショット・弾の削除処理です。タスクシステムでは、このようなキャラクターの削除をタスクの削除として扱います (Fig. 3-15)。

タスク削除の手順は、タスク生成の手順とはちょうど逆になります。ここではFig. 3-16のように自機・敵・弾という3つのタスクがあるときに、敵タスクを削除することを考えてみます。

敵タスクを削除するには、敵タスクをアクティブタスクリストから取り除きます。これは削除するタスクの前後にある2つのタスクを連結する処理に相当します。

そして、削除したタスクをフリータスクリストに移動します (Fig. 3-17)。ポインタを操作する都合上、フリータスクリストの先頭 (ダミーの直後) に移動するのが簡単です。こ

の際、挿入位置の直前にあるフリータスク（ダミー）についてポインタを操作します。

続いて、削除したタスクの保持しているポインタを変更して、フリータスクリストを結合し直します（Fig. 3-18）。アクティブタスクリストの削除箇所の前後にあるタスクのなかにあるポインタも操作する必要があります。Fig. 3-18では、変更の必要があるポインタを網掛けと太枠で示しました。

List 3-8は、タスクを削除するプログラムです。処理関数の設定などがないので、タスクを生成するプログラム（List 3-6）よりも簡単になっています。

このようにタスクシステムでは、タスク用のメモリを新たに確保することなく、同じメモリを繰り返し利用します。タスクの生成と削除を繰り返しても、全体として使用するメモリの量は最初に確保したときから変化しません。そのため、メモリの断片化などを心配することなく、自由にタスクの生成と削除を繰り返すことができます。シューティングゲームのように、弾や敵などを無数に生成したり削除したりするゲームでは、この性質がとても重要です。

**Fig. 3-15　キャラクターの削除＝タスクの削除**

Fig. 3-16 タスクを削除する前の状態
アクティブタスクリスト
ActiveTask（ダミー）
処理関数：—
前のタスク：タスク2
次のタスク：タスク0
ワークエリア
タスク0（自機）
処理関数：FuncMyShip
前のタスク：ActiveTask
次のタスク：タスク1
ワークエリア
タスク1（敵）
処理関数：FuncEnemy
前のタスク：タスク0
次のタスク：タスク2
ワークエリア
タスク2（弾）
処理関数：FuncBullet
前のタスク：タスク1
次のタスク：ActiveTask
ワークエリア
フリータスクリスト
FreeTask（ダミー）
処理関数：—
前のタスク：—
次のタスク：フリータスク1
ワークエリア
フリータスク1
処理関数：—
前のタスク：—
次のタスク：フリータスク2
ワークエリア
フリータスク2
処理関数：—
前のタスク：—
次のタスク：FreeTask
ワークエリア

Fig. 3-17 削除するタスクの移動

アクティブタスクリスト
ActiveTask（ダミー）
処理関数：—
前のタスク：タスク2
次のタスク：タスク0
ワークエリア
タスク0（自機）
処理関数：FuncMyShip
前のタスク：ActiveTask
次のタスク：タスク1
ワークエリア
タスク2（弾）
処理関数：FuncBullet
前のタスク：タスク1
次のタスク：ActiveTask
ワークエリア
削除するタスクを取り出し、
フリータスクリストに移動する
フリータスクリスト
FreeTask（ダミー）
処理関数：—
前のタスク：—
次のタスク：フリータスク1
ワークエリア
タスク1（敵）
処理関数：FuncEnemy
前のタスク：タスク0
次のタスク：タスク2
ワークエリア
フリータスク1
処理関数：—
前のタスク：—
次のタスク：フリータスク2
ワークエリア
フリータスク2
処理関数：—
前のタスク：—
次のタスク：FreeTask
ワークエリア

**Fig. 3-18** 削除が完了した状態

**List 3-8** タスクを削除する（Task1.cpp）

```
// タスクの削除
void DeleteTask(TASK* task) {

    // アクティブタスクリストからタスクを削除する
    task->Prev->Next=task->Next;
    task->Next->Prev=task->Prev;

    // 削除したタスクをフリータスクリストに挿入する
    task->Next=FreeTask->Next;
    FreeTask->Next=task;
}
```

# 処理関数の変更

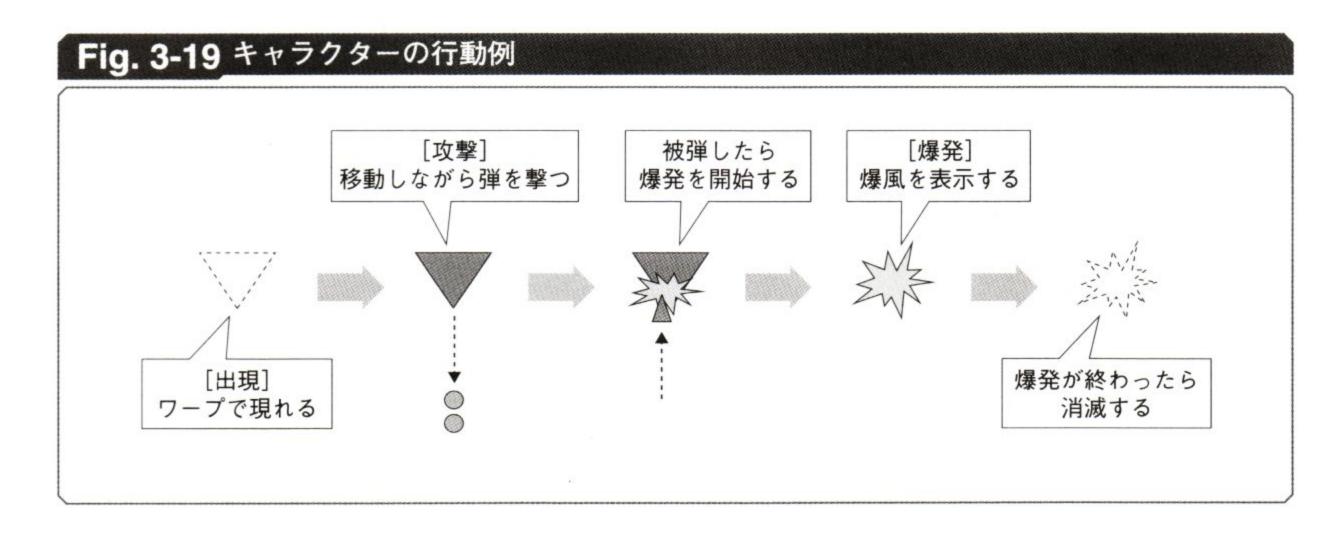
タスクは処理関数へのポインタを持っています。したがって、この関数ポインタを変更すれば、タスクの処理内容を自由に変えることができます。これをタスクチェンジと呼ぶこともあります。

ゲームに登場するキャラクターは、時間とともに次々と行動を変えます。例えば、Fig. 3-19のような敵について考えてみます。

この敵はワープで画面に出現します。出現したら攻撃を開始し、移動しながら弾を撃ちます。そして自機のショットを被弾すると爆発し、爆発のアニメーションを表示した後に消滅します。

この敵の行動は次のような3段階に整理できます。

- 出現（ワープで現れる）
- 攻撃（移動と弾の発射）
- 爆発（アニメーションの表示）

**Fig. 3-19 キャラクターの行動例**

このように状況に応じて行動を変えるキャラクターのプログラムは、処理関数ポインタの入れ替えを使って実現することができます（Fig. 3-20）。

タスクの生成時には、最初の処理関数として出現用の関数（Fig. 3-20ではEnemyAppear）を設定します。そして出現の動作が終わったら、処理関数を攻撃用の関数（EnemyAttack）に入れ替えます。

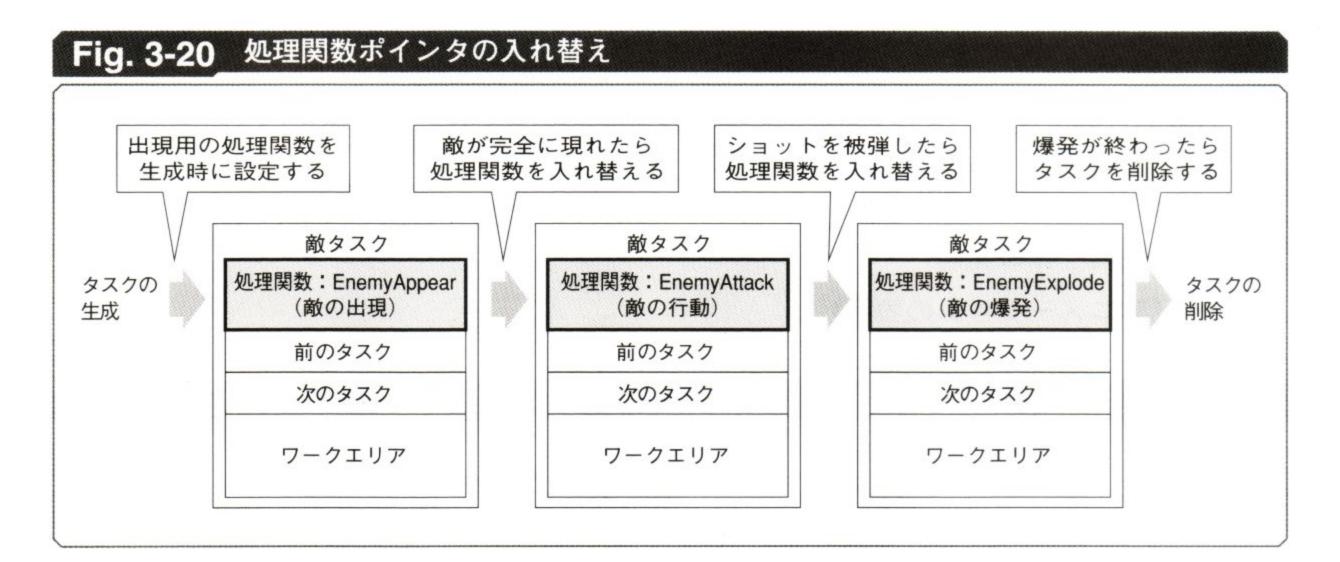
もしも攻撃中にショットを被弾したら、処理関数を爆発用の関数（EnemyExplode）に入れ替えます。そして爆発のアニメーションを表示し、爆発が終わったらタスクを削除します。

処理関数を入れ替えるときには、ワークエリアの内容は次の処理関数にそのまま引き継ぎます。新しい処理関数はワークエリアに残された座標などの情報を活用できるので、1つのキャラクターのふるまいを変更する場合には好都合です。

なお、処理関数を入れ替えて敵を爆発させる方法の他に、敵と同じ位置に新しい「爆発タスク」を生成する方法もあります。どちらも結果はほとんど同じなので、場面に応じて

使いやすい方法を選ぶとよいでしょう。次章以降では、タスクを生成することによって爆発を表現しています。

List 3-9は、処理関数を変更するプログラムの例です。

**Fig. 3-20 処理関数ポインタの入れ替え**

**List 3-9 処理関数を変更する（Task1.cpp）**

```
// 敵のワークエリア構造体
struct ENEMY_WORK {

    // 座標と速度
    float X, Y, VX, VY;

    // 耐久力とタイマー
    int Vit, Timer;
};

// 敵の処理関数（出現）
// 最初に設定される処理関数
void EnemyAppear(TASK* task) {

    // 出現の処理を行う（詳細は省略）

    // 一定時間が経過したら
    // 攻撃を開始するために処理関数を入れ替える
    ENEMY_WORK* work=(ENEMY_WORK*)task->Work;
    work->Timer++;
    if (work->Timer==6) task->Func=EnemyAttack;
}

// 敵の処理関数（攻撃）
```

```
void EnemyAttack(TASK* task) {

     // 攻撃の処理を行う(詳細は省略)

     // ここでは耐久力が0以下になったとき破壊されたとする
     // 破壊されたら爆発を開始するために処理関数を入れ替える
     ENEMY_WORK* work=(ENEMY_WORK*)task->Work;
     if (work->Vit<=0) {
          work->Timer=0;
          task->Func=EnemyExplode;
     }
}

// 敵の処理関数(爆発)
void EnemyExplode(TASK* task) {

     // 爆発の処理を行う(詳細は省略)

     // 一定時間が経過したらタスクを削除する
     ENEMY_WORK* work=(ENEMY_WORK*)task->Work;
     work->Timer++;
     if (work->Timer==12) DeleteTask(task);
}
```

## タスクの生成とワークエリアの初期化

タスクの生成時には、ワークエリアの初期化もあわせて行う必要があります。例えば、新しく敵を登場させるときには、敵タスクを生成し、敵の座標や速度などを設定しなければなりません。

こういったタスク生成とワークエリア初期化の処理は、「敵を作る関数」や「弾を作る関数」といった関数にまとめるとすっきりします。

例えば、敵を生成する関数には、敵を出現させる座標などの引数が必要です。関数はタスクを生成し、ワークエリアのポインタを敵ワークエリア構造体のポインタにキャストした後、構造体のメンバに座標や速度といった初期値を設定します。

List 3-10は、敵を作る関数の例です。

**List 3-10 タスクの生成とワークエリアの初期設定（Task1.cpp）**

```
// 敵の生成
// xとyは敵を出現させる座標
void CreateEnemy(float x, float y) {

    // タスクを生成する
    TASK* task=CreateTask(EnemyAppear);
    if (!task) return;

    // ワークエリアに初期値を設定する
    assert(sizeof(ENEMY_WORK)<=WORK_SIZE);
    ENEMY_WORK* work=(ENEMY_WORK*)task->Work;
    work->X=x;
    work->Y=y;
    work->VX=0;
    work->VY=0.1f;
    work->Vit=5;
    work->Timer=0;
}
```

## 弾やショットの生成

List 3-10と同じ方法で、弾やショットなどを生成する関数を作ることもできます。例えば弾には、狙い撃ち弾や誘導弾などいろいろな種類があるので、種類別に関数を用意しておくと便利です（弾の種類については後述します）。

また、n-way弾や円形弾のように、複数の弾から構成された弾幕を生成する関数を作ることもできます。例えばn-way弾ならば、

- 弾幕を飛ばす方向
- 弾の数
- 弾と弾との間隔

などを引数で指定するような関数を作ればよいでしょう。こういった関数を用意しておけば、関数を一度呼び出すだけで簡単に複雑な弾幕を生成することができます。

さまざまな弾やショットの初期設定と移動の方法については、書籍『シューティングゲームアルゴリズムマニアックス』をご覧いただければ幸いです。同書では、初期化処理と移動処理のそれぞれについてソースコードを掲載しています。この初期化処理をワークエリアの初期設定処理とし、移動処理をタスクの処理関数とすれば、掲載ソースコードをほぼそのままタスクシステムと組み合わせることができます。

# 当たり判定処理

ゲームのジャンルを問わず、ゲームプログラムには当たり判定処理が欠かせません。例えばシューティングゲームの場合には、

・自機と敵
・自機と弾
・敵とショット

といった組み合わせの当たり判定処理が必要です。ただし、自機と敵との当たり判定を取らないゲームもあります。一方、

・自機とショット
・敵と弾
・自機と自機
・敵と敵
・ショットとショット
・弾と弾

などについては当たり判定処理を行わないのが一般的です(Fig. 3-21)。ゲームによっては、自機と自機との当たり判定を取ることなどもあります。当たり判定の取り方によってゲーム性が大きく変わってくるので、面白く遊べるよういろいろと工夫するとよいでしょう。

ここでは、タスクシステムに当たり判定処理を組み込む方法の一例を紹介します。基本的な仕組みはFig. 3-22のとおりです。

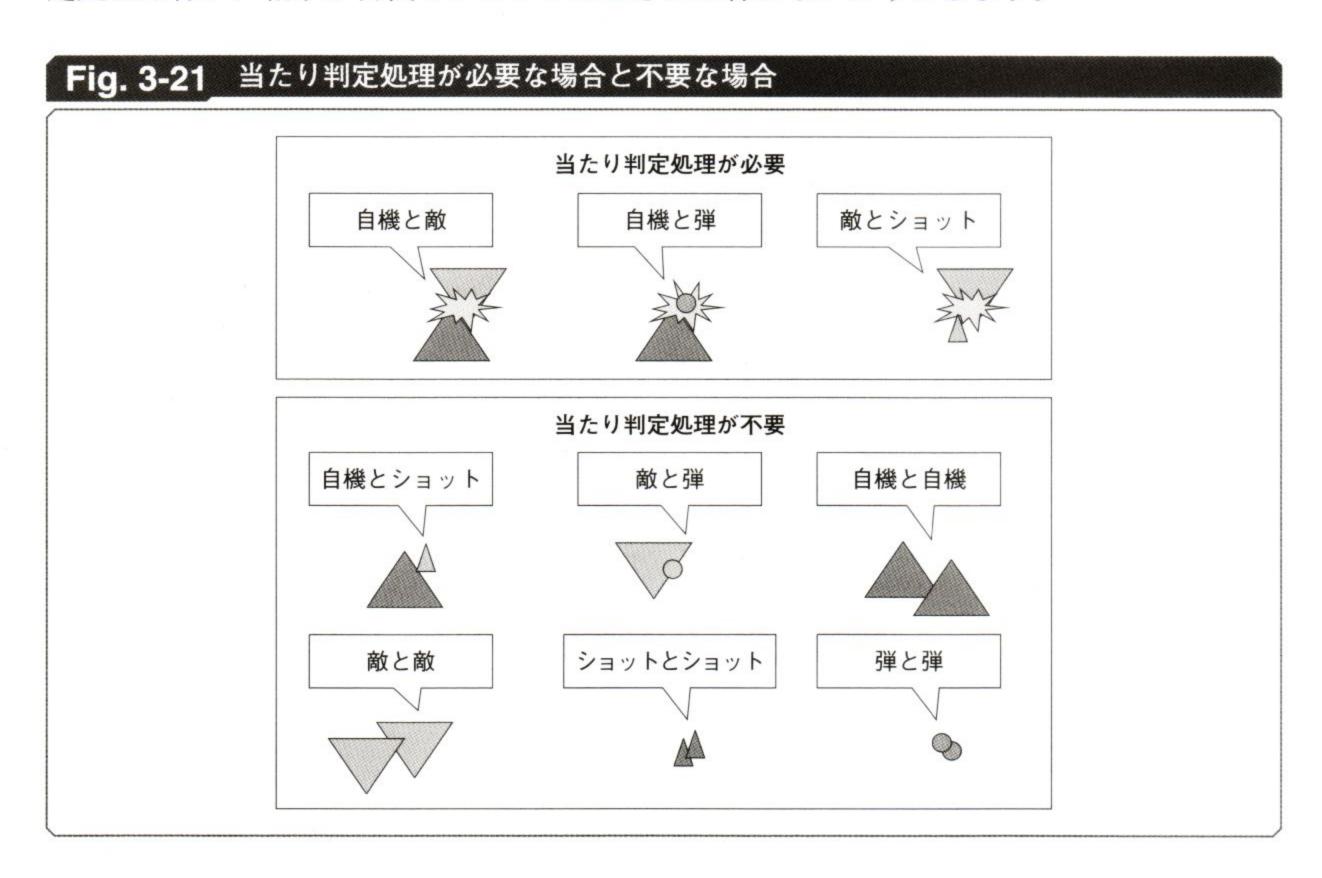
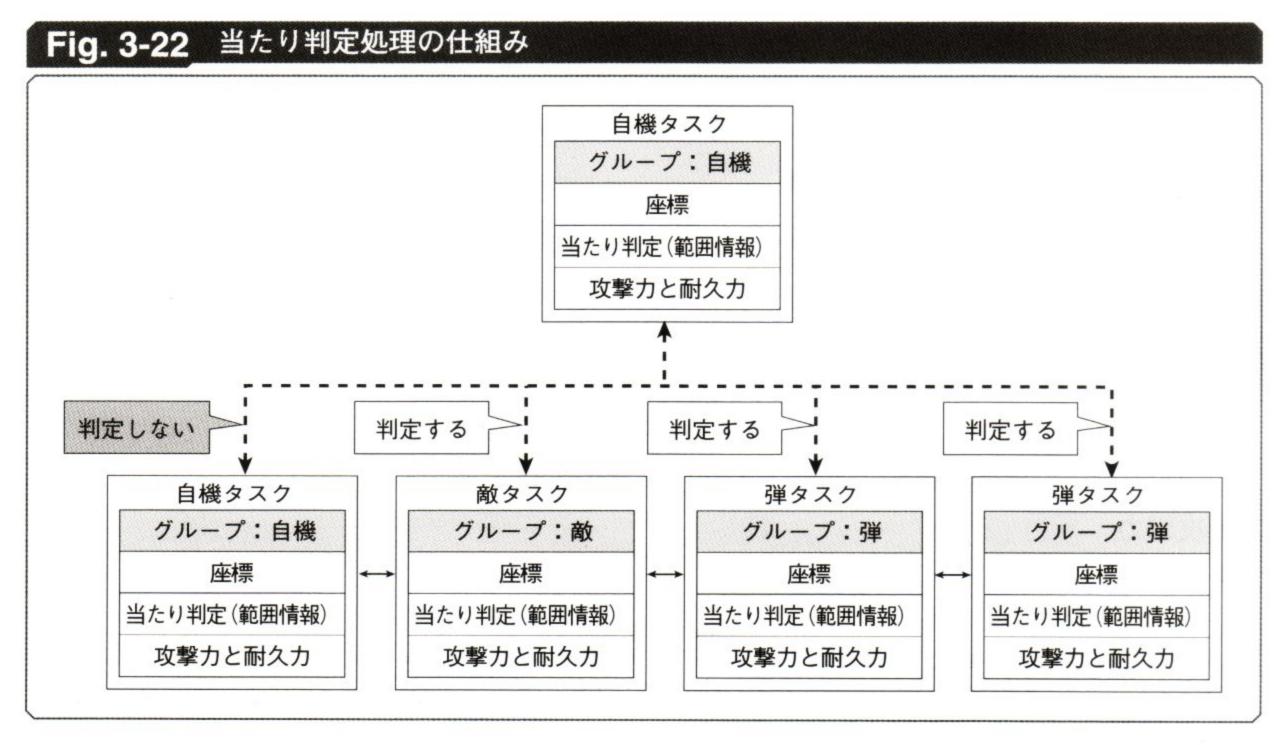
Fig. 3-22は、自機と敵、および自機と弾の当たり判定処理を行う場合です。各タスクには、

・グループ
・座標
・当たり判定(矩形。左端・右端・上端・下端の座標で表現)
・攻撃力
・耐久力

といった情報を持たせておきます。そして、自機タスクとアクティブタスクリストの全タスクとの間で当たり判定処理を行います。

この際にグループを調べることによって、必要な組み合わせでだけ当たり判定処理を行います。例えば自機タスクの場合には、相手のグループが敵または弾のときだけ当たり判定処理を行い、相手が自機やショットのときには行わないようにします。

**Fig. 3-21 当たり判定処理が必要な場合と不要な場合**

**Fig. 3-22 当たり判定処理の仕組み**

当たり判定の条件についてはChapter 1（P. 8）で解説しています。当たり判定の形状を表す座標データは、構造体などで表現するとよいでしょう。また、グループ・座標・攻撃力・耐久力といった情報を保持する変数も必要です。

当たり判定処理の後に耐久力が0以下になった場合には、タスクを削除します。実際のゲームではいきなりタスクを削除するのではなく、処理関数を変更して爆発などを表示してからタスクを削除することが多いでしょう。

List 3-11は、タスクシステムに当たり判定処理を組み込んだ例です。ここでは自機と敵、自機と弾、敵とショットとの間で当たり判定処理を行います。また、タスクシステム自体に当たり判定処理を組み込んでいますが、組み込まない方法もあります。Chapter 5では、当たり判定処理をタスクシステム自体には含めずに、タスククラスの派生クラスで実装する方法を紹介します（P. 160）。

**List 3-11　当たり判定処理（Task2.cpp）**

```
// 当たり判定用の構造体
struct HIT {
    float Left, Right, Top, Bottom;
};

// 当たり判定用のグループ分け
enum GROUP {
    GR_MYSHIP, GR_SHOT, GR_ENEMY, GR_BULLET
};

// タスクの構造体
struct TASK {

    // 処理関数へのポインタ
    void (*Func)(TASK* task);

    // 前後のタスクへのポインタ
    TASK* Prev;
    TASK* Next;

    // グループ
    GROUP Group;

    // 座標
    float X, Y;

    // 当たり判定
    HIT Hit;
```

```
    // 攻撃力と耐久力
    int Str, Vit;

    // ワークエリア
    char Work[WORK_SIZE];
};

// 当たり判定処理：
// task0とgroupに属する全タスクとの間で判定を行い、
// 双方の攻撃力で相手側の耐久力を減らす
void CheckHit(TASK* task0, GROUP group) {

    // task0の当たり判定の座標を求める
    float
        l0=task0->X+task0->Hit.Left,
        r0=task0->X+task0->Hit.Right,
        t0=task0->Y+task0->Hit.Top,
        b0=task0->Y+task0->Hit.Bottom;

    // すべてのタスクとの間で判定を行う
    for (TASK *task1=ActiveTask->Next, *next;
        next=task1->Next, task1!=ActiveTask; task1=next)
    {
        // task1が指定のグループではない場合には判定しない
        if (task1->Group!=group) continue;

        // task1の当たり判定の座標を求める
        float
            l1=task1->X+task1->Hit.Left,
            r1=task1->X+task1->Hit.Right,
            t1=task1->Y+task1->Hit.Top,
            b1=task1->Y+task1->Hit.Bottom;

        // task0とtask1との間で当たり判定処理を行い、
        // 当たった場合には双方の攻撃力で相手の耐久力を削る
        if (l0<r1 && l1<r0 && t0<b1 && t1<b0) {
            task0->Vit-=task1->Str;
            task1->Vit-=task0->Str;
        }
    }
}

// 自機の処理関数
void FuncMyShip(TASK* task) {
```

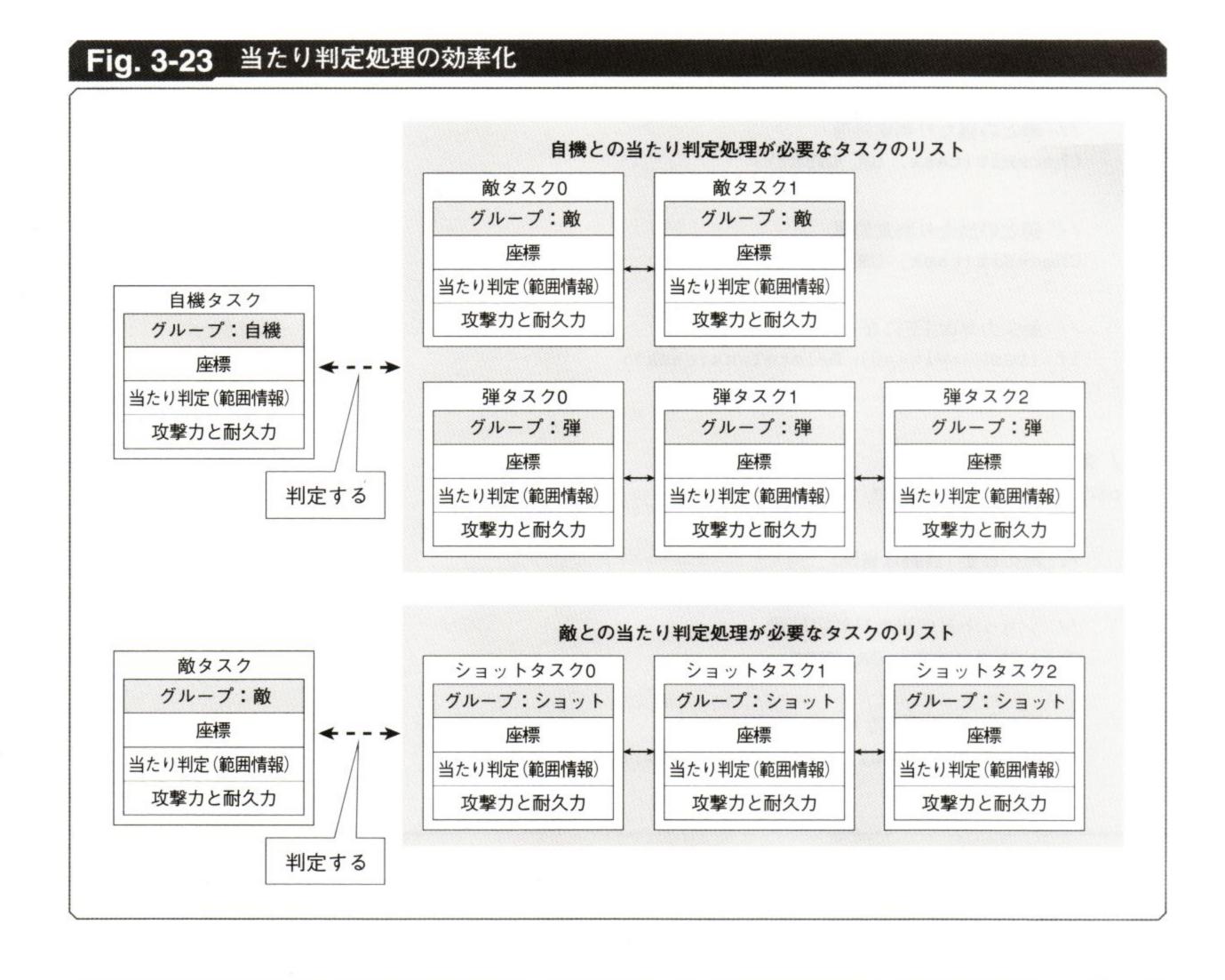
```
    // 自機の移動(詳細は省略)

    // 敵との当たり判定処理
    CheckHit(task, GR_ENEMY);

    // 弾との当たり判定処理
    CheckHit(task, GR_BULLET);

    // 耐久力が0以下になったら消滅する
    if (task->Vit<=0) DeleteTask(task);
}

// 敵の処理関数
void FuncEnemy(TASK* task) {

    // 敵の移動(詳細は省略)

    // ショットとの当たり判定処理
    CheckHit(task, GR_SHOT);

    // 耐久力が0以下になったら消滅する
    if (task->Vit<=0) DeleteTask(task);
}
```

## 当たり判定処理の効率化

ここで説明した当たり判定処理には少し効率の悪い点があります。それは、当たり判定処理が必要か不要かにかかわらず、全タスクに対するループ処理を行う点です。当たり判定処理が不要なタスクもチェックして、その所属するグループを調べるコストを払っています。

当たり判定処理を効率化する方法の1つは、タスクリストを敵や弾といったグループごとに分けておくことです(Fig. 3-23)。この場合、自機タスクは敵タスクリストや弾タスクリストとだけ判定処理を行います。また、敵タスクはショットタスクリストとだけ判定処理を行います。なお、Chapter 4以降では、敵や弾といった種類別にタスクリストを分けています。

グループ分けしたタスクリストは、順番に実行していきます。「自機タスクリストを動かす→敵タスクリストを動かす→弾タスクリストを動かす→…」といった感じです。

**Fig. 3-23 当たり判定処理の効率化**

## 移動と描画の処理関数を分ける

今までのプログラム例では、各タスク内にある処理関数は1つだけでした。この場合には、1つの処理関数のなかで移動と描画との両方を行います。

しかし実際のゲームプログラムでは、移動と描画で処理を分けておくと便利なことがあります。例えば次のような場合です。

- 処理速度が追いつかないので、描画を何回か省略して移動だけを行いたい
- 重ね合わせやアルファ合成の都合上、移動と違う順序で描画を行いたい

移動と描画を分けるには、処理関数へのポインタを移動用と描画用の2つに分けておきます。タスクを生成する際には、移動用と描画用の両方のポインタに値を設定します。

全タスクを移動するには、全タスクに関して移動用の処理関数を呼び出します。一方、

全タスクを描画するには、全タスクに関して描画用の処理関数を呼び出します。

自機・敵・ショット・弾などの指定したグループのタスクだけを描画することもできます。それには、全タスクのグループを調べて、特定のグループに属するタスクに関してだけ描画用の処理関数を呼び出します。

List 3-12は、移動と描画とを分けた場合のプログラム例です。

**List 3-12 移動と描画とを分けたタスク構造体（Task3.cpp）**

```
// 処理関数の型宣言
struct TASK;
typedef void (*FUNC)(TASK* task);

// タスクの構造体
struct TASK {

    // 処理関数へのポインタ（移動と描画）
    FUNC FuncMove;
    FUNC FuncDraw;

    // 前後のタスクへのポインタ
    TASK* Prev;
    TASK* Next;

    // グループ
    GROUP Group;

    // ワークエリア
    char Work[WORK_SIZE];
};

// タスクの生成
TASK* CreateTask(FUNC func_move, FUNC func_draw, GROUP group) {

    // ...（中略）...

}

// 全タスクの移動
void MoveTask() {
    for (TASK *task=ActiveTask->Next, *next;
        next=task->Next, task!=ActiveTask; task=next)

        // 移動用の処理関数を呼び出す
        (*task->FuncMove)(task);
}
```

```
// 全タスクの描画
void DrawTask() {
    for (TASK *task=ActiveTask->Next, *next;
        next=task->Next, task!=ActiveTask; task=next)

        // 描画用の処理関数を呼び出す
        (*task->FuncDraw)(task);
}

// 指定したグループのタスクを描画する
void DrawTask(GROUP group) {
    for (TASK *task=ActiveTask->Next, *next;
        next=task->Next, task!=ActiveTask; task=next)

        // 特定のグループに属するタスクに関してだけ
        // 描画用の処理関数を呼び出す
        if (task->Group==group) (*task->FuncDraw)(task);
    }
}
```

## タスクシステムが持つその他の機能

今までに解説した機能だけでも、十分に実用的なタスクシステムを構築することができます。一方、タスクシステムには、次のような機能もつけ加えることができます。これらの機能は必須ではないので、必要が生じた時点で追加すればよいでしょう。

### タスクの優先度

各タスクに優先度を設定して、タスクの実行順序を制御する機能です。

### 全タスクの削除

場面転換などで多くのタスクを一気に消したいときに便利な機能です。

### 親子タスク

依存関係のあるタスク間に親子関係を設けて管理する機能です。

### タスクの検索

指定した条件に合致するタスクを探し出す機能です。

### デバッグとチューニングの支援

変数値や実行時間といった情報を出力することによって、デバッグやチューニングを助ける機能です。

## メモリ管理に連結リストを使う理由

本書のタスクシステムでは、アクティブタスクリストとフリータスクリストという2つの連結リストを使っています。これは複雑に見えるかもしれませんが、実はメモリ管理を効率化することが目的です。

とはいえ、連結リストを使うと、なんだか煩雑に感じるかもしれません。もっと簡単な方法について検討してみましょう。

### 連結リストを使わないタスクシステム

連結リストを使わないタスクシステムとして、Fig. 3-24のような例を想定してみます。このタスクシステムは、連結リストを使うタスクシステムに比べると、かなりわかりやすい構造をしています。しかし実は、タスクシステムを作成する際に陥りやすい問題を抱えています。

この方法では、フリータスクリストは用意しません。あらかじめ固定長のメモリ領域を確保しておき、新しいタスクを作るときには領域の先頭からタスク用のメモリを確保していきます。

タスクの大きさは不ぞろいとします。タスクの派生クラスを定義し、派生クラスのオブジェクト生成に必要なメモリを確保していくと、大きさは自然と不ぞろいになります。

タスクを消去したら消去ずみのマークをつけて、メモリ上には残しておきます。そして、消去ずみタスクがある場合でも、新しいタスクは常に空き領域の先頭から確保します。タスクの大きさが不ぞろいなので、新しいタスクのサイズに合う消去ずみタスクの領域を見つけるには手間がかかるからです。

さて、タスクを次々に生成していくと、いつかは空き領域が足りなくなります。空き領域が不足したら、Fig. 3-25のような方法で空き領域を確保します。

空き領域が不足した場合には、消去ずみタスクの領域を詰めます。これは、消去ずみタ

スクよりも後方にあるタスク用のメモリを、前方にコピーすることによって行います。この操作をメモリ領域の先頭から末尾まで行えば、空き領域が1つにまとまり、再びタスクを生成できるようになります。

**Fig. 3-24 連結リストを使わないタスクシステム**

1. 固定長のメモリ領域を確保する

| 空き領域 |
|---|

2. 空き領域の先頭からタスク用のメモリを確保する

| タスク0 | タスク1 | タスク2 | 空き領域 |
|---|---|---|---|

3. タスクを消去したら消去ずみタスクにする（メモリ上には残る）

| タスク0 | 消去ずみタスク | タスク2 | 空き領域 |
|---|---|---|---|

4. 消去ずみタスクがある場合でも、新しいタスク用のメモリは空き領域の先頭から確保する

| タスク0 | 消去ずみタスク | タスク2 | タスク3 | 空き領域 |
|---|---|---|---|---|

**Fig. 3-25 メモリ領域を詰める操作**

1. 空き領域が不足した場合

| タスク0 | 消去ずみタスク | タスク2 | タスク3 | 消去ずみタスク | タスク5 | 空き領域 |
|---|---|---|---|---|---|---|

2. タスク用のメモリを前方にコピーして隙間なく詰めることにより、空き領域を確保する

| タスク0 | タスク2 | タスク3 | タスク5 | 空き領域 |
|---|---|---|---|---|

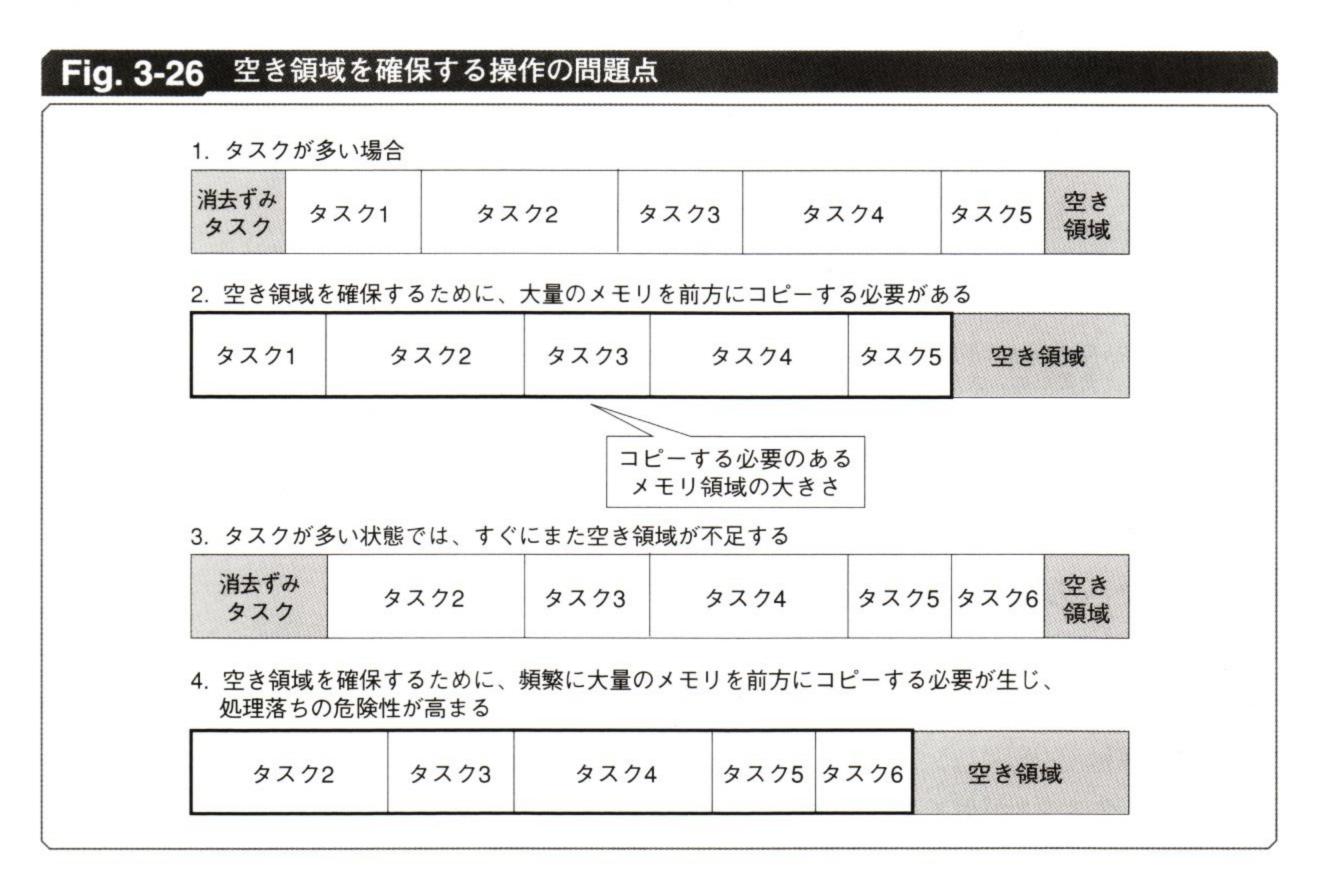
## 3つの問題点

さて、この方法は簡単でわかりやすく思えます。実装も比較的容易でしょう。また、各タスクが必要としているだけのメモリを割り当てるため、メモリ領域も無駄なく使えるような印象があります。

ところがこのタスクシステム、実は次のような問題点を抱えています。

## 空き領域を確保する操作が重い

空き領域が不足したときには、未消去のタスク用のメモリを前方にコピーすることによって、メモリ領域の先頭から隙間なくタスクを詰めます。未消去のタスクが少なければよいのですが、タスクが多い場合には大量のメモリをコピーしなければなりません（Fig. 3-26）。

**Fig. 3-26　空き領域を確保する操作の問題点**

最近はマシンが速くなったのでメモリのコピーも高速ですが、それでも大量のメモリをコピーすればそれなりに時間がかかります。携帯ゲーム機などの比較的性能が低いマシンでは、問題はより大きくなります。

特に、タスクの数が多く、メモリの使用率が高い状態では、非常に好ましくない動作をするおそれがあります。空き領域を確保しても、すぐにまた空き領域を確保する必要が生じるため、頻繁に大量のメモリを前方にコピーする結果になります。処理時間がかかり、処理落ちやフレーム落ちが発生する危険性が高まります。

## 効率的に動かすためには過剰にメモリを消費する

この方法では、多くのタスクがメモリに配置されていてメモリの使用率が高いときには、頻繁に空き領域が不足します。すると、空き領域を確保するための重い処理を繰り返すことになります。これを防ぐためには、タスク群が実際に使用するメモリ量に比べて大きな

メモリ領域を確保することによって、メモリ使用率を低くしなければなりません(Fig. 3-27)。

これでは過剰にメモリを消費することになります。この方法は各タスクが必要としているだけのメモリを確保し、タスクを空き領域の先頭から詰めるため、一見メモリ領域を無駄なく使えるように思えます。ところが実際には逆に、効率的に動かすためにはメモリ領域を過剰に大きく確保しておかなければならないのです。

**Fig. 3-27** 過剰にメモリを消費する問題点

1. メモリ使用率が高いと、頻繁に空き領域が不足する

| タスク0 | タスク1 | タスク2 | 空き領域 |
|---|---|---|---|

2. 空き領域の不足を防止するには、メモリ使用率を低くする必要がある
そのため、大きな領域を確保することになる

| タスク0 | タスク1 | タスク2 | 空き領域 |
|---|---|---|---|

### タスクへのポインタが使えない

空き領域を確保する操作を行うと、タスクのメモリ上における位置、つまりアドレスが変化します。もし、あるタスクがポインタを使って他のタスクを参照していると、問題が起きます(Fig. 3-28)。

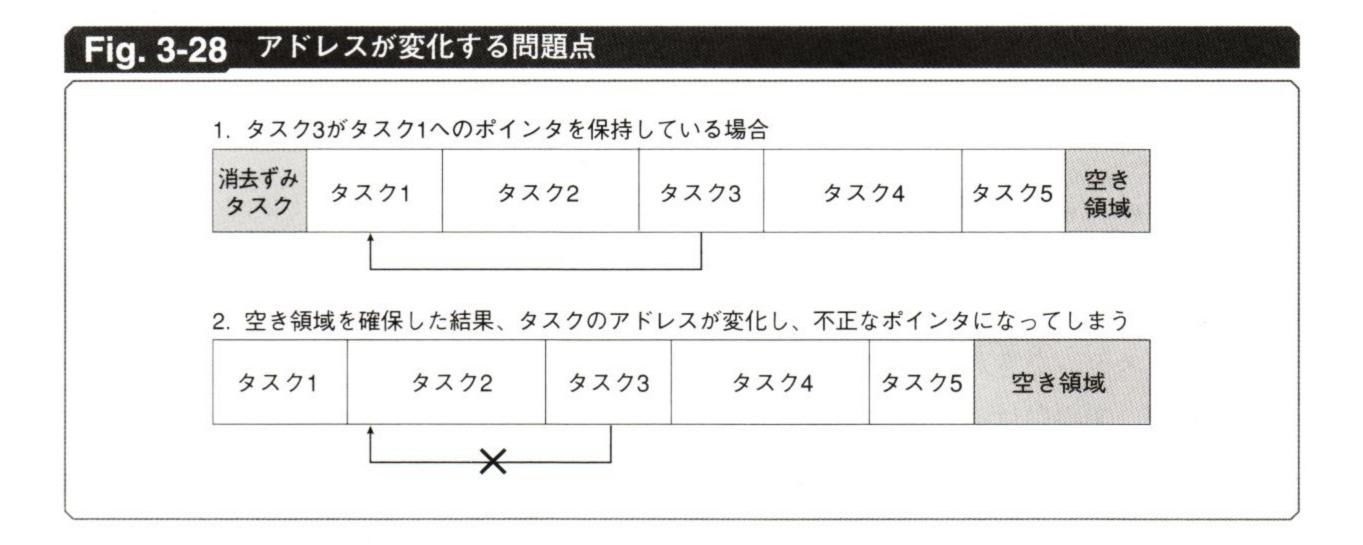
例えば、タスク3がポインタを使ってタスク1を参照しているとします。このように他のタスクをポインタで参照する状況としては、自機から発射されたビームが自機を参照するとか(P. 137)、分裂したボスが相互を参照するといった場合があります(P. 294)。

ここで空き領域が不足すると、空き領域を確保するためにタスク用のメモリを前方にコピーします。するとタスクのアドレスが変化するため、タスク3が持つポインタはタスク1を正しく指さずに、予期せぬタスクを指してしまうことになります。これは不正なポインタなので、もはや使用することができません。

※

この方法を利用するときには、以上のような問題点があることを認識しておく必要があります。連結リストを用いた方法では、これらの問題は発生しません。

なお、メモリのコピーを使って空き領域を確保する方法は、一種のガベージコレクションだといえます。この方法はわかりやすいのですが、ガベージコレクションをゲームに採用すべきかどうかは、検討の余地があります(P. 385)。

Fig. 3-28 アドレスが変化する問題点

## タスクのサイズが著しく異なる場合

連結リストを使用した本書のタスクシステムでは、タスクリストごとにタスクのサイズが固定されています。タスクのサイズは、そのタスクリストに属するさまざまな種類のタスクのなかで最大のものに合わせます。

そのため、サイズの小さい種類のタスクでは、タスクの末尾に未使用のメモリ領域が生じます。これは無駄に見えるかもしれません。しかし、タスクを固定長にすることによって、実行時におけるメモリの確保と解放のコストが大幅に削減できるのです。多くのケースでは、タスクを固定長にしてメモリの確保・解放の効率化を図る方が有利でしょう。

タスクのサイズが著しく異なる場合、例えば20バイトを必要とするタスクと10000バイトを必要とするタスクが混在するような場合には、タスクリストを分けるとよいでしょう。ほぼ同じサイズのタスクごとにタスクリストを分けておけば、未使用のメモリ領域を小さくできます。

多くのゲームでは、自機タスクリスト・弾タスクリスト・敵タスクリストといった具合に、キャラクターの種類に応じてタスクリストを分けることになります。この場合、タスクごとのサイズの違いはそれほど気にする必要はありません。例えば、弾には方向弾・狙い撃ち弾・誘導弾といったさまざまなものがありますが、タスクのサイズはほとんど同じです。特にタスクのサイズが著しく異なる場合だけ、タスクリストを分けるとよいでしょう。

# クラスを使ったタスクシステム

これまではクラスを使わずにタスクシステムを書いてきました。しかし、C++を使うからにはクラスを使いたい、と思うのが人情です。それに、タスクの「関数ポインタ＋ワークエリア」という構造は「仮想関数を持つクラス」によく似ているので、クラスを使った方がタスクシステムをきれいに書けそうな気がします。

ところが、実際にはそう簡単ではありません。タスクの処理関数ポインタと仮想関数は一見同じもののようですが、性質はまるで違います。例えば、処理関数ポインタは自由に入れ替えられるものですが、仮想関数を入れ替えるにはオブジェクトを丸ごと入れ替える必要があります。

しかし、多少の工夫をすればタスクシステムをクラス化することができます。ここでは、newとdeleteのオーバーロードを使った方法を紹介します。

new演算子とdelete演算子をオーバーロードすることによって、タスクのメモリを確保する際に、フリータスクリストから確保するようにします。そして、仮想関数を使ってさまざまなクラスのメンバ関数を呼び分けます。

この方法には、List 3-9 (P. 70) のような処理関数の変更ができないという制限があります。しかし、処理関数の入れ替えは以下のような方法で代替できます。

- 処理内容が変わるところで別のタスクを生成する
- タスクの状態に応じてswitch文で処理を分岐させる
- 仮想関数のなかから処理関数を関数ポインタを使って呼び出す

## C言語とC++のどちらを使うか

タスクシステムはC言語の構造体と関数ポインタを使用しても実現できますが、C++が普及した現在ではクラスを利用して実現することが多くなっています。クラスを使った場合には、以下のような利点があります。

- 構造体を使った場合よりも名前空間をすっきりと整理できる
- タスクの生成をコンストラクタにまとめることができる
- タスクの削除をデストラクタにまとめることができる
- 構造体と関数を組み合わせるよりもデータと処理を一体化できる

C言語よりもC++を使った方が、おそらくプログラムは簡潔になります。C++に慣れているのならば、C++で作ったタスクシステムの方が使いやすいと感じるでしょう。

## タスクリストを分ける

さて、これまでに解説したタスクシステムはFig. 3-29のような構成でした。自機・弾・敵といったすべてのタスクを、アクティブタスクリストとフリータスクリストという2つのリストで管理しています。アクティブタスクは活動中のタスクを、フリータスクは未割り当てのタスクを表します。Prevは前のタスク、Nextは次のタスクです。また、ワークエリアは各タスクが自由に使用できるメモリ領域を示します。

Fig. 3-29の方式では、さまざまな種類のタスクを単一の系統のリストで管理しますが、これには少し問題があります。まず、タスクの種類ごとに出現数の上限が設けられていないため、例えば弾を出現させすぎると自機のショットが出現しなくなるといった現象が起こります。

また、特定のタスクに対して繰り返し処理を行う場合に、処理の無駄が多いのも問題です。例えば弾に対してだけ処理を行う場合にも、タスクリスト内にあらゆるタスクが混在しているため、自機や敵といった無関係なタスクを避けながら処理を進めなければなりません。これに関しては当たり判定処理の効率化に付随して述べました（P. 77）。

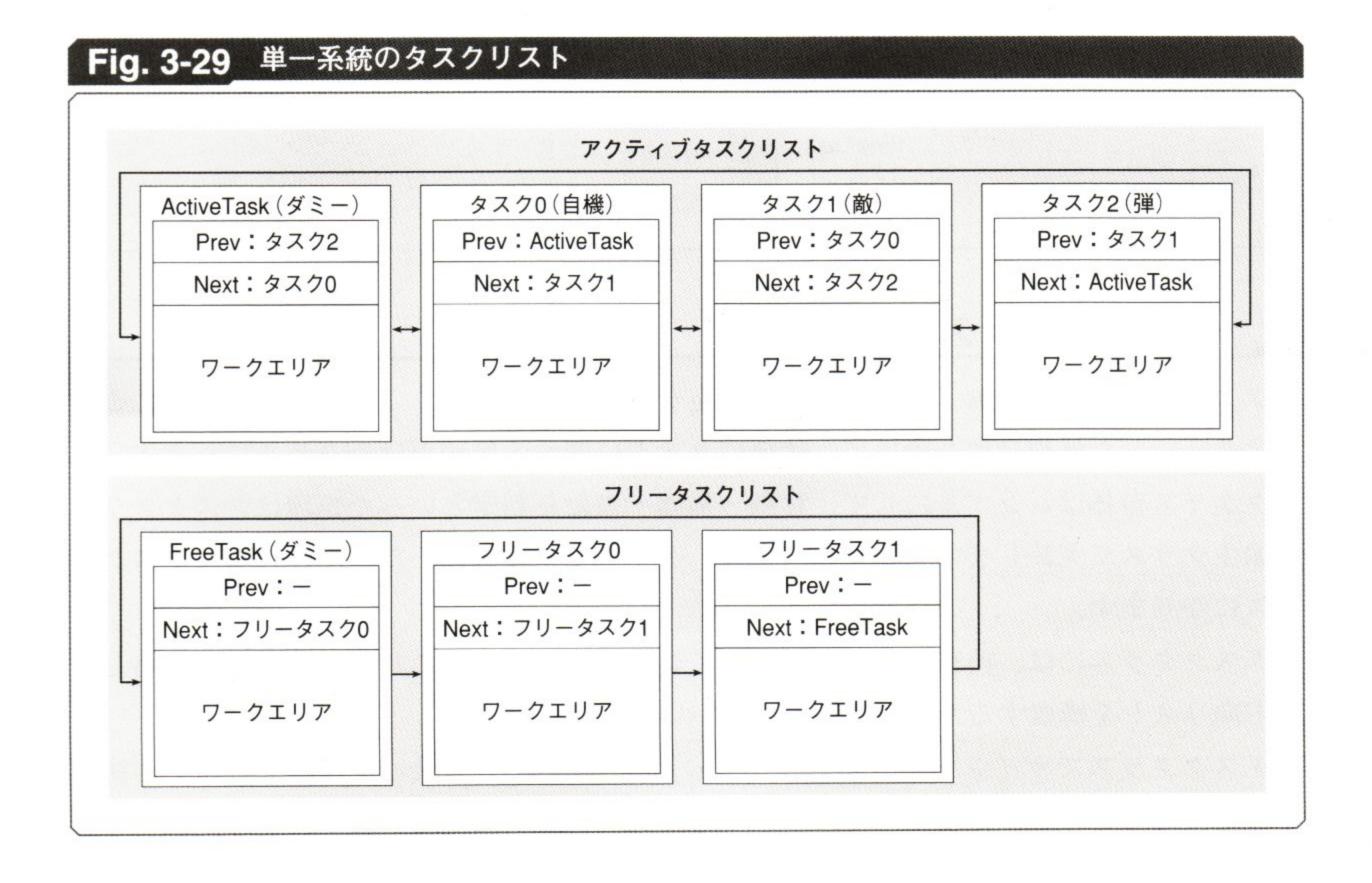
**Fig. 3-29** 単一系統のタスクリスト

こういった問題を回避するには、タスクリストを複数の系統に分ける方法があります。Fig. 3-30のように、複数のタスクリストを用意して、それぞれにアクティブタスクリストとフリータスクリストを設けます。そして、自機、弾、敵といったタスクの種類別にタスクリストを使い分ければ、前述のような問題は解消されます。

Fig. 3-30の方針に基づいて作成したタスクとタスクリストのクラスについて、次に説明します。

**Fig. 3-30 複数系統のタスクリスト**

弾のタスクリスト

アクティブタスクリスト
ActiveTask（ダミー） ⇔ タスク0（方向弾0） ⇔ タスク1（方向弾1） ⇔ タスク2（狙い撃ち弾0）

フリータスクリスト
FreeTask（ダミー） → フリータスク0 → フリータスク1

敵のタスクリスト

アクティブタスクリスト
ActiveTask（ダミー） ⇔ タスク0（赤身0） ⇔ タスク1（赤身1） ⇔ タスク2（玉子0）

フリータスクリスト
FreeTask（ダミー） → フリータスク0

## タスクをクラスで表現する

タスククラスには移動・描画・当たり判定などの処理を含めることもできますが、ここではそういった処理は取り除いて、純粋にタスクに関する処理だけをクラスにします。タスククラス自体はシンプルにして、移動・描画・当たり判定といった処理はタスククラスの派生クラスで実装します。こうした方が、ライブラリ化に向いた汎用性のあるタスククラスになります。

タスククラスには、前後のタスクへのポインタが必要です。これは、タスクを連結して双方向リストを構成するためです。

タスククラスでポイントになるのは、newとdeleteの実装方法です。new演算子が使用されているときには、適切なフリータスクリストからタスクを1つ取り出して、適切なア

クティブタスクリストに加えます。delete演算子が使用されたら、そのタスクを適切なフリータスクリストに移動します。そのため、タスククラスでは以下のように、タスクが属すべきタスクリストをnewやdeleteに渡す必要があります。

```
bullet=new(BulletList) CDirBullet();
delete(BulletList) bullet;
```

タスクリストを指定しないnewやdeleteは、正しく処理を行うことができません。そこで、以下のようなnewとdeleteをprivateメンバとして定義しておきます。privateメンバなので、これらのnewやdeleteはクラスの外から呼び出せなくなります。

```
void* operator new(size_t t) {}
void operator delete(void* p) {}
```

このようなnewとdeleteを定義した状態で、タスクリストを指定しない以下のような呼び出しを行うと、コンパイルエラーが出るようになります。実行前にエラーを検知できるので、誤った呼び出しを効率的に修正することができます。

```
new CDirBullet();
delete bullet;
```

さらに、以下のような関数をprotectedメンバとして用意します。これらはnewとdeleteに相当する処理を行う関数です。

```
static void* operator_new(size_t t, CTaskList* task_list);
static void operator_delete(void* p, CTaskList* task_list);
```

第2引数にはタスクリストを取ります。タスククラスの派生クラスでは、これらの関数を使ってnew演算子とdelete演算子をオーバーロードします。newやdeleteが使用されたら、これらの関数を呼び出すように、newとdeleteを定義するわけです。

このような方法を用いるのは派生クラスごとに異なるタスクリストを指定するためです。派生クラスにおけるnewとdeleteの定義については後述します(P. 93)。

それから、コンストラクタとデストラクタも必要です。コンストラクタはタスクリストを引数に取ります。こちらも処理の詳細は後述します(P. 93)。

List 3-13は、タスククラス(CTask)です。これ以降のソースコードは、付録CD-ROMの「LibGame」フォルダに収録しています。

**List 3-13　タスククラス（Task.h）**

```
class CTask {
friend CTaskList;
friend CTaskIter;

private:

    // タスクリストへのポインタ
    CTaskList* TaskList;

    // 前後のタスクへのポインタ
    CTask *Prev, *Next;

    // タスクリストを指定しないnewとdeleteが呼ばれたときに
    // コンパイルエラーを発生させるための関数
    void* operator new(size_t t) {}
    void operator delete(void* p) {}

protected:

    // newとdeleteの処理：
    // 適切なnew演算子、delete演算子をサブクラスで定義する
    static void* operator_new(size_t t, CTaskList* task_list);
    static void operator_delete(void* p, CTaskList* task_list);

public:

    // コンストラクタ、デストラクタ
    CTask(CTaskList* task_list);
    virtual ~CTask();
};
```

## タスクリストをクラスで表現する

タスクリストのクラスは、1組のアクティブタスクリストとフリータスクリストを管理します。また、タスクの最大サイズや最大数、フリータスクの数なども保持します。

List 3-14は、タスクリストのクラスです。

**List 3-14　タスクリストクラス（Task.h）**

```
class CTaskList {
friend CTask;
friend CTaskIter;
```

```
    // アクティブタスクリスト、フリータスクリスト
    CTask *ActiveTask, *FreeTask;

    // タスクの最大サイズ、タスクの最大数
    int MaxTaskSize, MaxNumTask;

    // フリータスクの数
    int NumFreeTask;

public:

    // コンストラクタ
    CTaskList(int max_task_size, int max_num_task);

    // タスクの数
    int GetNumFreeTask() { return NumFreeTask; }
    int GetNumActiveTask() { return MaxNumTask-NumFreeTask; }

    // 全タスクの消去
    void DeleteTask();
};
```

## タスクリストの初期化

前述のように、タスクリストにはアクティブタスクリストとフリータスクリストがあります。最初は、タスクリストをFig. 3-31のように初期化します。

各タスクリストには、ActiveTaskとFreeTaskというダミーのタスクがあります。これらのダミータスクは、タスクリストから削除されることがありません。ダミータスクを使うと、タスクリストが空のときと空でないときの処理を分ける必要がなくなるため、プログラムを簡潔にし、処理の効率化を図ることができます（P. 52）。

タスクリストを初期化するには、まずタスク用のメモリを確保します。必要なメモリ量は、使用する最大のタスク数にダミータスク数（2個）を加えた個数に、タスクの最大サイズを乗じたサイズです。

次に、アクティブタスクリストとフリータスクリストを初期化します。メンバ変数のPrevとNextを設定して、Fig. 3-31の状態を作り出します。

List 3-15は、タスクリストを初期化するプログラムです。

**Fig. 3-31 初期状態のタスクリスト**



**List 3-15 タスクリストの初期化**

```
CTaskList::CTaskList(int max_task_size, int max_num_task)
:   MaxTaskSize(max_task_size), MaxNumTask(max_num_task),
    NumFreeTask(max_num_task)
{
    // タスク初期化用のマクロ
    #define TASK(INDEX) ((CTask*)(buf+max_task_size*(INDEX)))

    // タスク用メモリの確保
    char* buf=new char[max_task_size*(max_num_task+2)];

    // アクティブタスクリストの初期化
    ActiveTask=TASK(0);
    ActiveTask->Prev=ActiveTask->Next=ActiveTask;

    // フリータスクリストの初期化
    FreeTask=TASK(1);
    for (int i=1; i<max_num_task+1; i++) {
        TASK(i)->Next=TASK(i+1);
    }
    TASK(max_num_task+1)->Next=FreeTask;
}
```

# タスクの生成

タスクの生成はFig. 3-32のように行います。フリータスクリストからタスクを1個取り出し、アクティブタスクリストに追加します。タスクを削除するには、Fig. 3-33のようにタスクをアクティブタスクリストからフリータスクリストに戻します。

## タスクを生成する方法

タスクの生成は、newの処理とコンストラクタの処理に分けます。タスクの生成にはメモリの確保とメンバ変数の初期化が必要で、newだけではメンバの初期化はできないからです。

newのなかで、確保したメモリ領域の先頭をキャストしてthisポインタとして使えば、newのなかでもメンバ変数の初期化ができるように思えます。しかし、これは危険な方法です。タスククラスが多重継承された場合には、newの確保したメモリの先頭とthisが一致しない場合があります。そのため、newでメンバの初期化を行うと、誤ったメモリ領域を操作する危険性があります。これが初期化処理をnewとコンストラクタに分けなければならない理由です。

なお、タスククラスではnew演算子をオーバーロードしません。かわりに、newに相当する処理を行う関数（operator_new）を用意します。派生クラスでは、この関数を利用してnew演算子をオーバーロードします。これは、派生クラスごとにタスクリストを変えられるようにするためです。

さて、通常のnewではヒープからメモリを確保しますが、このタスクシステムの場合にはフリータスクリストからフリータスクを1個取得します。この際には、フリータスクがまだ存在するかどうかと、確保するサイズがタスク1個あたりの最大サイズを超えていないかどうかをチェックします。フリータスクが取得できたら、ポインタを設定して、アクティブタスクリストにタスクを追加します。

このタスクシステムでは、newとコンストラクタの両方にタスクリストへのポインタを与える必要があります。両方にタスクリストを指定するのは面倒そうですが、これはタスクの派生クラスでnewとコンストラクタを定義すれば解決できます。具体的なプログラムについてはChapter 4で紹介します（P. 122）。

グローバル変数やstatic変数を使えば、newからコンストラクタにタスクリストへのポインタを渡すことも可能です。しかし、マルチスレッドを使用する場合には、クリティカルセクションなどを利用してnewとコンストラクタの処理を不可分にする必要があります。

**Fig. 3-32　タスクの生成**

アクティブタスクリスト

ActiveTask（ダミー）
Prev：タスク2
Next：タスク0
ワークエリア

タスク0（自機）
Prev：ActiveTask
Next：タスク1
ワークエリア

タスク1（敵）
Prev：タスク0
Next：タスク2
ワークエリア

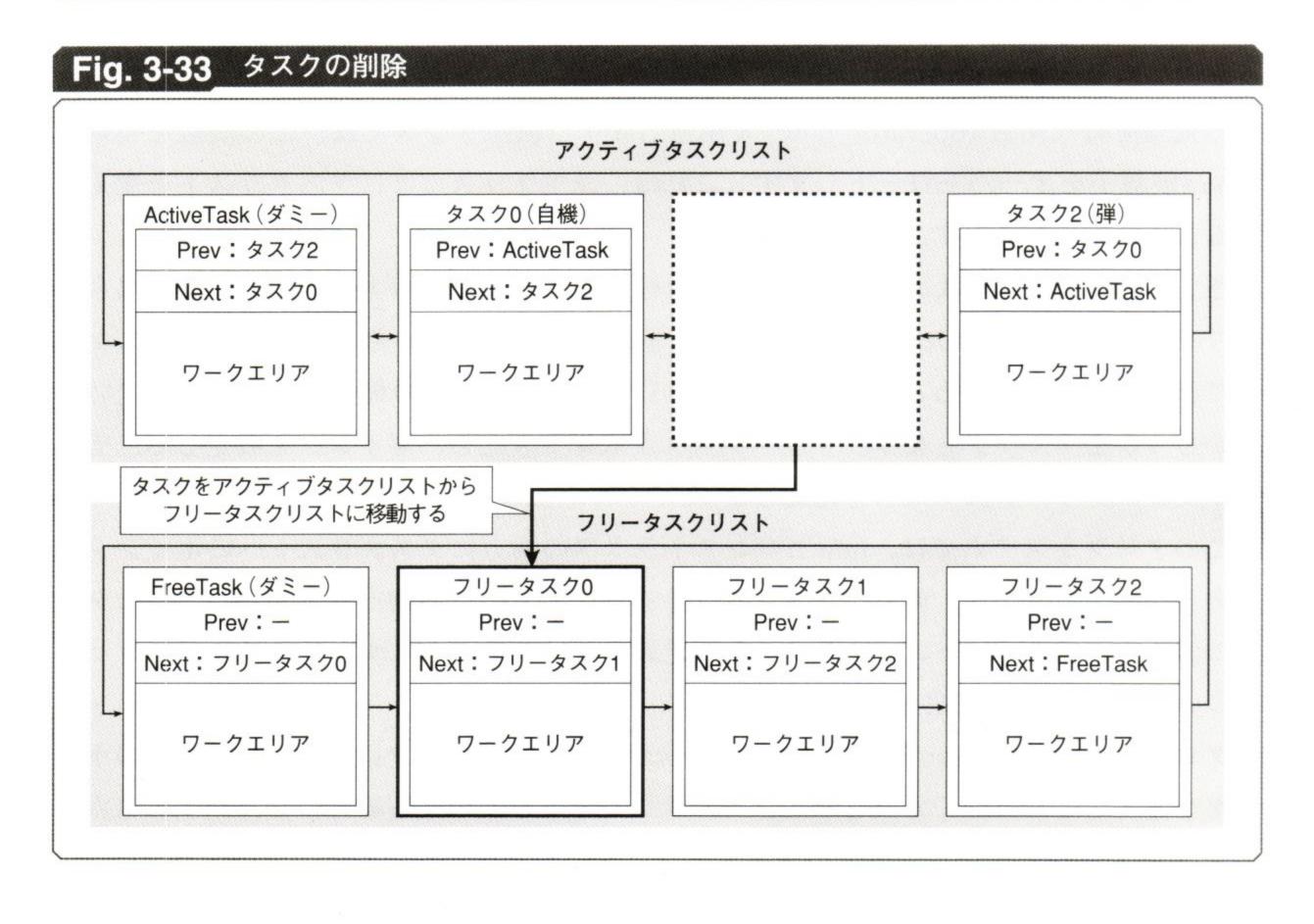
タスク2（弾）
Prev：タスク1
Next：ActiveTask
ワークエリア

タスクをフリータスクリストから
アクティブタスクリストに移動する

フリータスクリスト

FreeTask（ダミー）
Prev：ー
Next：フリータスク1
ワークエリア

フリータスク1
Prev：ー
Next：FreeTask
ワークエリア

**Fig. 3-33　タスクの削除**

あるタスクのnewとコンストラクタを実行している間に、他のタスクのnewとコンストラクタの実行が割り込むと、グローバル変数やstatic変数が書き換わることにより、タスクリストが不正な状態になることがあるからです。

結論としては、マルチスレッドの場合にも正しく動作させることを考えれば、newとコンストラクタの両方にタスクリストを引数として渡すのがシンプルで安全な方法です。

List 3-16は、タスクを生成するプログラムです。

**List 3-16　タスクの生成（Task.cpp）**

```
// newに相当する処理を行う関数
void* CTask::operator_new(size_t t, CTaskList* task_list) {

    // 確保するサイズがタスクの最大サイズを超えたらエラー
    assert(t<=(size_t)task_list->MaxTaskSize);

    // フリータスクがないときはNULLを返す
    if (task_list->NumFreeTask<=0) return NULL;

    // フリータスクを1個取り出す
    CTask* task=task_list->FreeTask->Next;
    task_list->FreeTask->Next=task->Next;
    task_list->NumFreeTask--;

    // コンストラクタへ
    return task;
}

// コンストラクタ
CTask::CTask(CTaskList* task_list)
:   TaskList(task_list)
{
    // タスクをアクティブタスクリストに挿入する
    Prev=task_list->ActiveTask->Prev;
    Next=task_list->ActiveTask;
    Prev->Next=Next->Prev=this;
}
```

# タスクの削除

タスクを削除するには、前述のようにタスクをアクティブタスクリストからフリータスクリストに戻します。生成と同様の理由で、削除の処理もデストラクタとdeleteに分けます。

メンバ変数の操作はデストラクタで行います。デストラクタではポインタを操作して、タスクをアクティブタスクリストから削除します。

タスククラスではdelete演算子をオーバーロードせず、かわりにdeleteに相当する処理を行う関数（operator_delete）を用意します。派生クラスでは、この関数を利用してdelete演算子をオーバーロードします。これはnew演算子と同じく、delete演算子の引数にタスクリストを指定するためです。

deleteの処理では、タスクをフリータスクリストに追加します。フリータスク数が最大タスク数を超えていないかどうかのチェックも行います。

このチェックは、deleteの重複呼び出しを検出するためです。deleteの重複呼び出しなどを行うと、結果としてフリータスク数が最大タスク数を超えることがよくあります。完璧なチェック方法ではありませんが、deleteの重複呼び出しは不正なメモリアクセスにつながるので、できるだけ誤りを検出できるようにこのチェックを入れています。

なお、deleteの処理では、新たにフリータスクになったメモリ領域をタスククラス（CTask）にキャストして使います。この領域は必ずタスククラスのサイズよりも大きいので、このキャストは安全です。

List 3-17は、タスクを削除するプログラムです。

**List 3-17　タスクの削除（Task.cpp）**

```
// デストラクタ
CTask::~CTask() {

    // タスクをアクティブタスクリストから削除する
    Prev->Next=Next;
    Next->Prev=Prev;
}

// deleteに相当する処理を行う関数
void CTask::operator_delete(void* p, CTaskList* task_list) {

    // タスク
```

```
    CTask* task=(CTask*)p;

    // タスクをフリータスクリストに挿入する
    task->Next=task_list->FreeTask->Next;
    task_list->FreeTask->Next=task;
    task_list->NumFreeTask++;

    // フリータスク数が最大タスク数を超えたらエラー
    assert(task_list->NumFreeTask<=task_list->MaxNumTask);
}
```

## タスクに対する繰り返し処理

タスクシステムでは、アクティブタスクリスト上のすべてのタスクに対して、なんらかの処理を繰り返し行うことがよくあります。例えば、すべてのタスクに対して移動・描画・削除などといった処理を行う場合です。シューティングゲームの場合、弾や敵などを移動したり描画したりする際に、このような繰り返し処理を行います。

タスクに対する繰り返し処理を簡単に記述する方法としては、マクロを使用する方法や、イテレータ（繰り返し処理を行うためのクラス）を使用する方法などがあります。本書ではイテレータを使うことにしました。イテレータ（iterator）というのは「反復子」や「繰り返し子」と呼ばれ、データ構造に対する繰り返し処理を抽象化した概念です。

イテレータというと、デザインパターンのIteratorパターンをご存じの方も多いでしょう。このタスクシステムにおけるイテレータも、部分的にIteratorパターンを利用しているといえます。

Iteratorパターンでは、さまざまなデータ構造に対する繰り返しのインタフェースを共通化するために、抽象クラスを使用します。このタスクシステムにおけるイテレータでは、繰り返し対象となるデータ構造がタスクリストだけなので抽象クラスを使う必要はありませんが、抽象クラスを導入すればIteratorパターンになります。

さて、ここで紹介するタスクイテレータは、タスクリスト上のタスクに対する繰り返し処理を行います。タスクイテレータは、繰り返し処理の対象となるタスクリストと、現在処理中のタスクを保持しています。

初期状態のタスクイテレータは、現在処理中のタスクとしてアクティブタスクリストの先頭にあるダミータスクを指しています。タスクを1つ処理するたびに、現在処理中のタスクを次のタスクに変更します。

タスクイテレータには、次のタスクを返す機能と、次のタスクがあるかどうかを調べる機能が必要です。また、直前に返したタスクを削除する機能も用意します。直前に返したタスクを削除するのは、現在処理中のタスクを削除するのに比べて安全で使いやすいからです。

List 3-18は、タスクイテレータのクラスです。

**List 3-18 タスクイテレータ**

```
// タスクイテレータ
class CTaskIter {
protected:

    // タスクリスト、タスク
    CTaskList* TaskList;
    CTask* Task;

public:

    // コンストラクタ
    inline CTaskIter(CTaskList* task_list)
    :  TaskList(task_list), Task(task_list->ActiveTask)
    {}

    // 次のタスクがあるときtrueを返す
    inline bool HasNext() {
        return Task->Next!=TaskList->ActiveTask;
    }

    // 次のタスクを返す
    inline CTask* Next() {
        return Task=Task->Next;
    }

    // 直前に返したタスクを削除する
    inline void Remove() {
        Task=Task->Prev;
        delete Task->Next;
    }
};
```

### タスクイテレータの使用例

タスクイテレータを使用すると、タスクに対する繰り返し処理を簡単に記述することができます。基本的な処理の流れは以下のとおりです。

① 次のタスクがあるかどうかを調べる
② 次のタスクがある場合には、次のタスクを取得する
③ 取得したタスクに対して移動や描画などの処理を行う
④ 必要があれば処理ずみのタスクを消去する

List 3-19は、タスクイテレータの簡単な使用例です。ここではアクティブリスト上の全タスクを削除します。より複雑な例はChapter 4で紹介します（P. 123）。

**List 3-19　全タスクの削除**

```
void CTaskList::DeleteTask() {
    for (CTaskIter i(this); i.HasNext(); i.Next(), i.Remove()) ;
}
```

## タスクの削除とイテレータ

タスクイテレータによる繰り返し処理とタスクの削除を同時に行う場合には、注意が必要です。Fig. 3-34のように、あるイテレータが処理中のタスクを削除してしまうと、イテレータの参照先がフリータスクに変わってしまい、誤動作します。

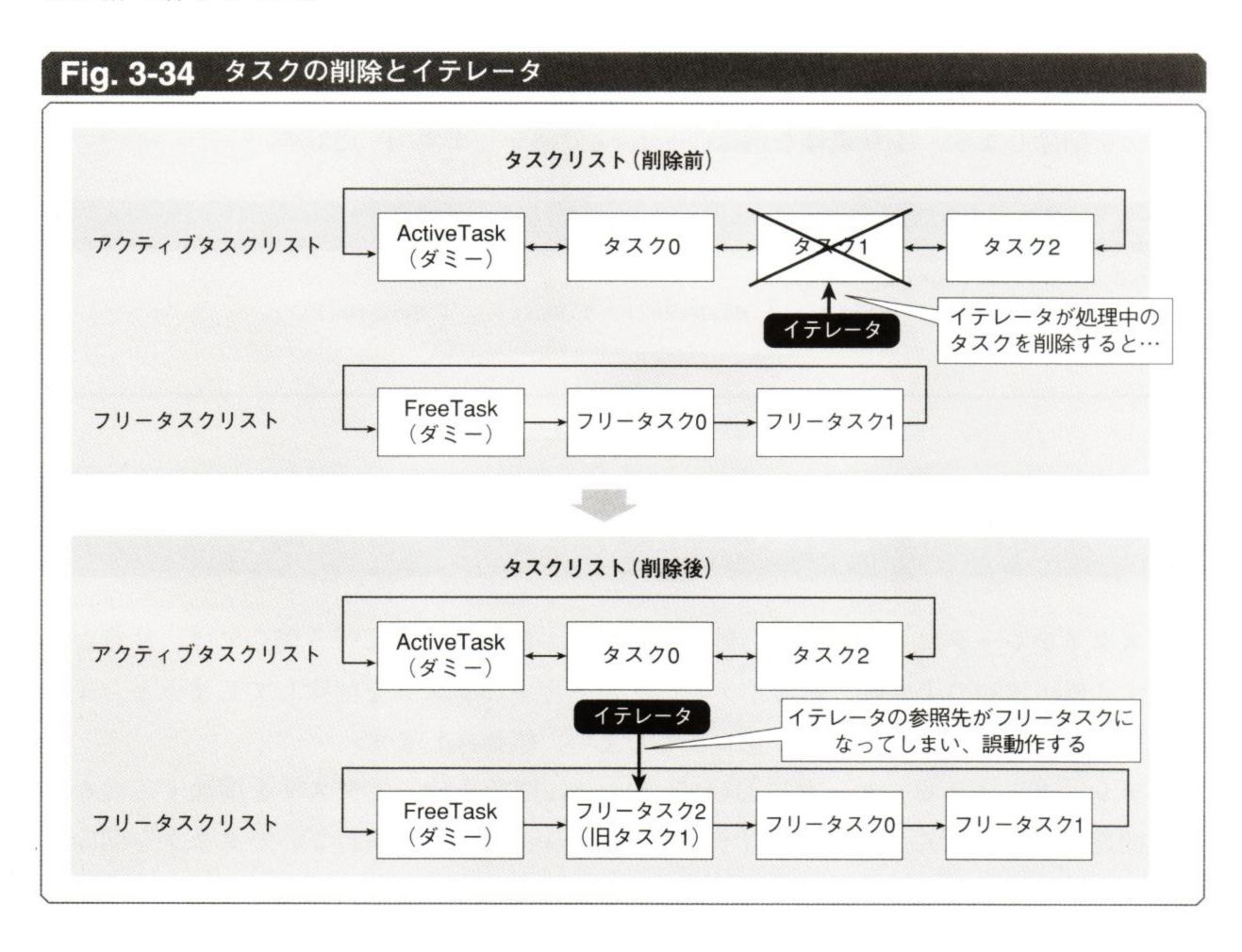
イテレータによる単一ループにおいて、Remove関数を使ってタスクを削除するのならば、問題は起きません。しかし、イテレータのRemove関数を使わないでタスクを削除したり、イテレータによる多重ループでタスクを削除したりすると、問題が発生する危険性があります。これを防止するには、イテレータが参照している可能性のあるタスクを、そのイテレータのRemove関数以外の方法で削除しないように、プログラマーが注意しなければなりません。

プログラマーの注意力に頼るのではなく、イテレータに誤動作防止機能を組み込む方法もあります。例えば、タスクリスト上にイテレータの配列などを用意して、イテレータを生成するたびに配列へ追加し、イテレータを削除するたびに配列から除去します。そして、タスクを削除する際にすべてのイテレータをチェックして、イテレータが処理中のタスク

が削除される場合には、問題が起きないようにイテレータの参照先を調整します。

この方法を使うと、プログラマーが特別な注意を払わなくても、タスクを安全に削除することができます。しかし、イテレータの生成と削除、そしてタスクの削除に関する処理は重くなります。プログラムもList 3-17ほどシンプルにはできません。

2つの方法を実装して検討したのですが、プログラマーがイテレータの扱いに注意したうえでプログラムを書けば、多くの場合には問題なくプログラムを書くことができそうでした。題材がゲームということもあるので、実行時の効率を重視して現状のシンプルな実装に落ち着きました。

**Fig. 3-34 タスクの削除とイテレータ**

# ゲームにおけるタスクシステムの活用

C++を使ってシューティングゲームを作ろうと思ったとき、おそらくいちばん気がかりなのはクラスの設計方法でしょう。設計の流儀は人それぞれでかまわないと思いますが、本書では本章で制作したタスクシステムをベースにして、ゲーム全体のクラス構成を設計

します。

## 自機・敵・弾などのクラス構成

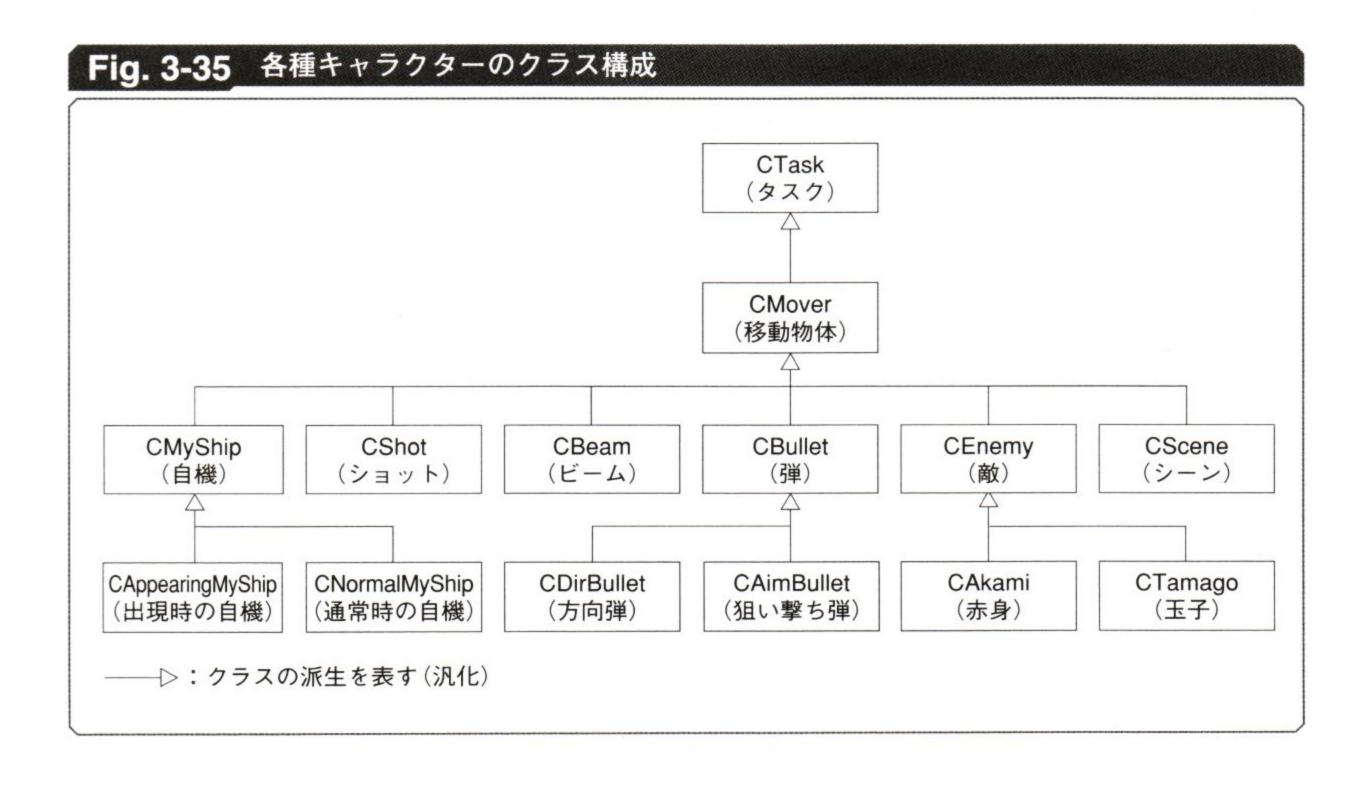
自機や敵といった各種キャラクターのクラスは、タスククラスのサブクラスとして作成します。Fig. 3-35は本書におけるクラス構成の例です。

ここではUML（Unified Modeling Language）の記法を用いました。矩形はクラスを表し、白抜きの三角形がついた矢印はクラスの派生を表します。

すべてのクラスはタスククラス（CTask）をスーパークラスとします。タスククラスから移動物体クラス（CMover）を派生させ、ここから各種キャラクターのクラスを派生させます。例えば、弾クラス（CBullet）や敵クラス（CEnemy）といった具合に、キャラクターの種類ごとに派生クラスを作成します。

弾クラスには弾に共通する性質、敵クラスには敵に共通する性質をまとめ、これらを基底クラスとして各種の弾や敵に対応する派生クラスを定義します。例えば、弾ならば方向弾クラス（CDirBullet）や狙い撃ち弾クラス（CAimBullet）、敵ならば赤身クラス（CAkami）や玉子クラス（CTamago）などのクラスを派生させます。

キャラクター以外に、ステージ・タイトル画面・ゲームオーバー画面などの処理についても同様に、クラスにまとめておきます。これらは、シーンクラス（CScene）から派生させることにしました。

**Fig. 3-35** 各種キャラクターのクラス構成

## プログラム全体の動き

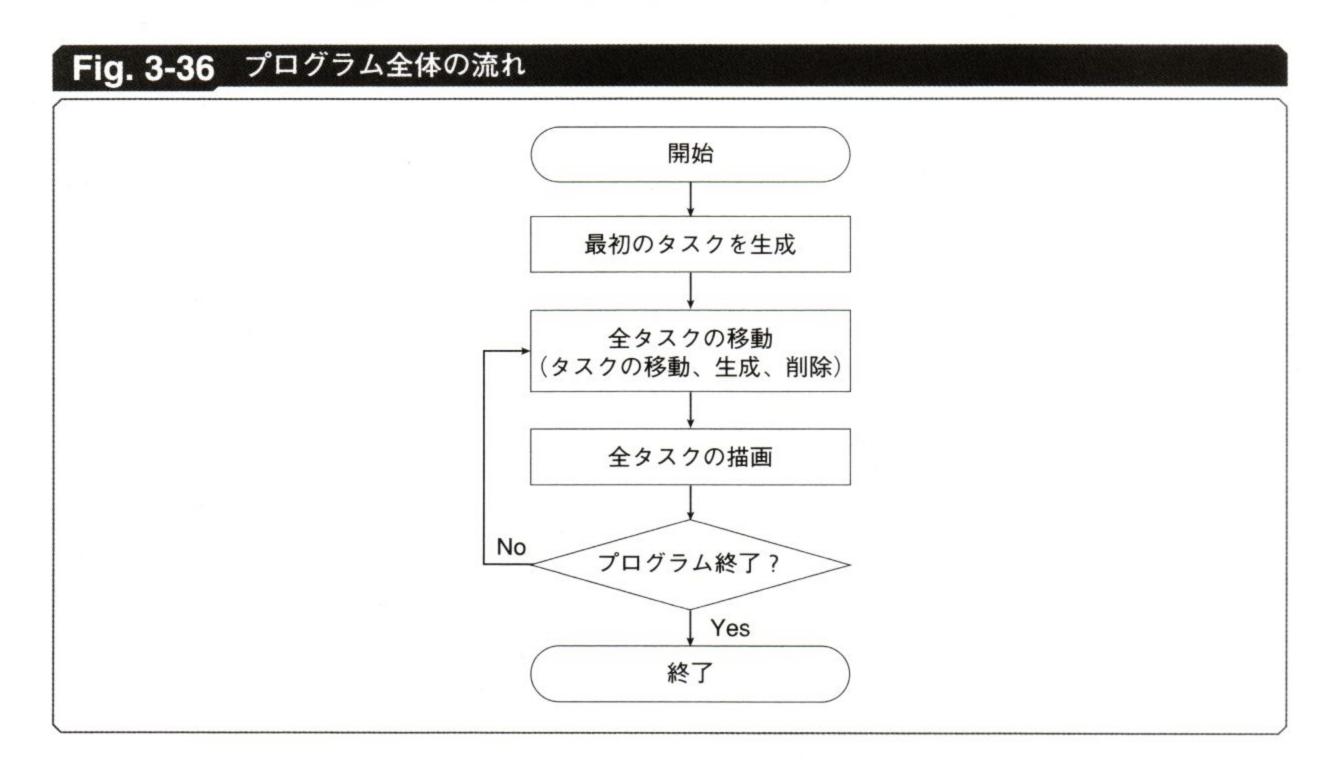
タスククラス（CTask）を中心にしてタスクリストを構成すること、そしてタスククラスの派生クラスとして各種キャラクターに相当するクラスを定義することを解説しました。次に、これらのクラスがどのように連携することによって、シューティングゲームの全体像が形作られるのかを説明します。

Fig. 3-36のように、プログラム全体の流れは非常にシンプルです。初期化を行い、最初のタスクを生成した後は、プログラム終了までタスクの移動と描画をひたすら繰り返します。

より具体的には、プログラムの初期化時に、自機や敵、弾などにグループ分けしたタスクリストを生成しておきます。あとはプログラムのメイン処理などで最初のタスクを生成し、そのタスクの処理にしたがって、新しいタスクを該当するタスクリストに追加していきます。あとは、タスクリストを順番に実行していきながら、タスクの処理とそれに伴う追加・削除を繰り返していきます。

この単純な仕組みから複雑なゲームが形作られる理由は、各タスクが次々に新しいタスクを生成することにあります。例えば、自機タスクはショットタスクやビームタスクを生成し、敵タスクは方向弾タスクや狙い撃ち弾タスクを生成します。そのため、最初はたった1個のタスクからスタートしても、ゲームが進行するうちに数多くのタスクが生成され、膨大な数のキャラクターが画面に登場することになるのです。

Fig. 3-37はタスクの生成関係を図にしたものです。最初にメインプログラム（CShtGame）

**Fig. 3-36** プログラム全体の流れ

が、タイトル画面（CTitle）を生成します。タイトル画面でゲームの開始が選択されると、今度はゲームのステージ（CStage）が生成されます。

ステージは自機（CMyShip）を生成し、赤身（CAkami）や玉子（CTamago）といった敵を生成します。自機はボタン操作に応じてショット（CShot）やビーム（CBeam）を生成し、敵は方向弾（CDirBullet）や狙い撃ち弾（CAimBullet）を生成します。

敵は破壊されると、敵の爆発（CEnemyCrash）を生成した後に、自らを削除します。自機が破壊された場合には、自機の爆発（CMyShipCrash）を生成します。自機の爆発はアニメーションの終了後に、残機がなければゲームオーバー画面（CGameOver）を生成します。ゲームオーバー画面は一定時間だけメッセージを表示した後に、タイトル画面（CTitle）を生成して自らを削除します。

このように、プログラム全体の流れは非常に単純ですが、多くのタスクを次々に生成することによって表現豊かなゲームを形作っています。あとはタスクのクラスを追加し、敵や弾、ステージなどのバリエーションを広げることによって、ゲームをどんどん発展させることができます。

**Fig. 3-37** タスクの生成関係

## >>Chapter 3のまとめ

本章ではタスクシステムについて解説しました。タスクシステムは処理関数ポインタとワークエリアを組み合わせたシンプルな仕掛けながら、ゲームプログラムの性質によく合っています。また、タスクごとに処理が明確に分かれているので、多人数でプログラムの開発を分担しやすいという利点もあります。例えば、あるプログラマーが自機タスクとショットタスクを作り、別のプログラマーが敵タスクと弾タスクを作るといった分担も比較的簡単にできます。

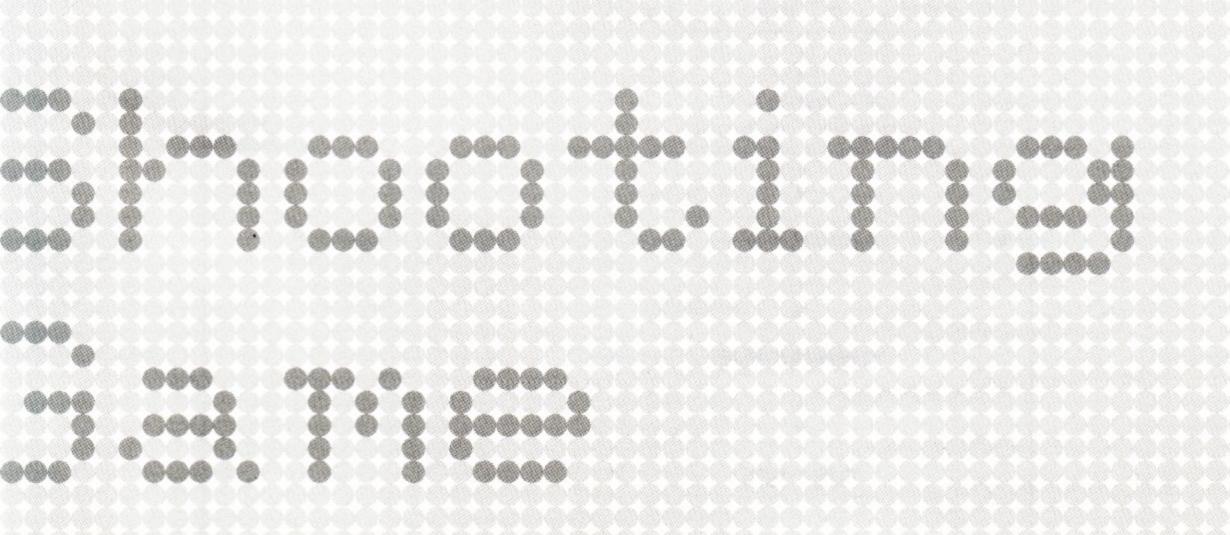
C++を使わずに、C言語だけで実現できることもタスクシステムの魅力です。さらにC++をうまく使えば、余分なキャストやポインタ操作を追い出し、データと処理を一体化することによって、より簡潔できれいなプログラムにすることができます。

メモリ管理もタスクシステムの重要な役割です。本書のタスクシステムではメモリ管理を効率よく行うために、連結リストを用いてタスクのメモリを管理します。タスクシステムの実装方法によっては、メモリ管理を効率よく行えない場合もあるので、方式をよく検討することが重要です。

次章からはいよいよシューティングゲームの制作に入ります。

# Chapter 04

# 自機

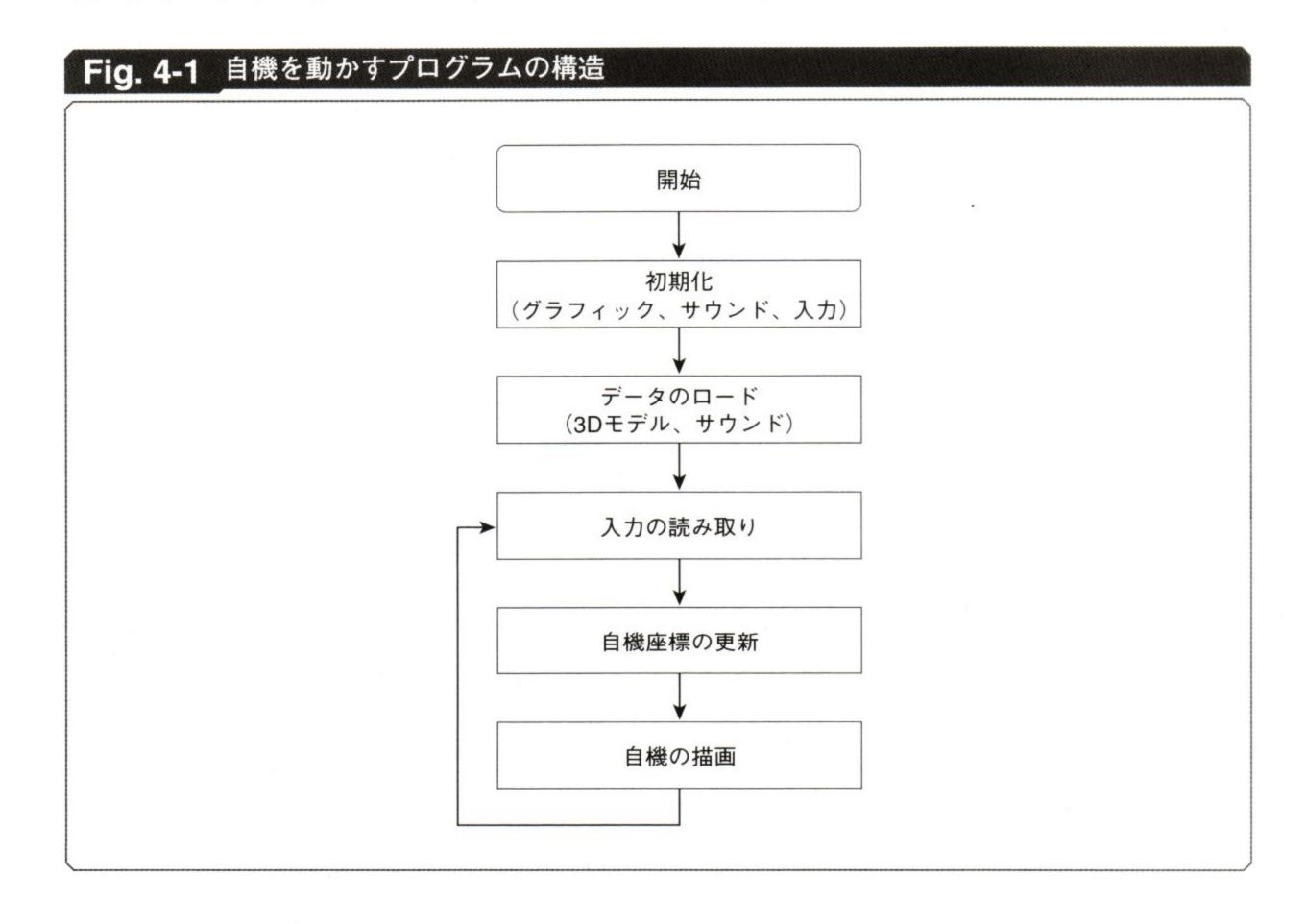
いよいよ、実際にシューティングゲームを作ります。自機や敵、弾、ステージなど、章ごとに要素を増やしていきましょう。
まずは、自機に関するプログラムからスタートします。自機の移動、ショットやビーム（レーザー）の発射などについて説明します。

# 自機を動かす

シューティングゲームをどこから作り始めるかは、人それぞれでかまわないと思います。でも、もしもどこから着手してよいのかわからなかったら、とりあえず自機を動かすだけの簡単なプログラムから作り始めるのがお勧めです。

最初はゲームになっていなくてもかまいません。重要なのは、ゲームプログラムの核となる部分を作り、とりあえず自機を動かすところまで到達することです。自機を動かすプログラムができれば、それを土台にして弾や敵などのプログラムを追加し、だんだん本格的なゲームに仕上げていくことができます。

自機を動かすプログラムは、Fig. 4-1のような構造をしています。C++とDirectXを使ったゲームプログラムは複雑で難解だという印象がありますが、実はFig. 4-1からもわかるように、ゲームの本質的な部分に関する構造は意外に単純です。

初期化、データの読み込み、入力の読み取り、自機の描画といった処理には、Chapter 2で作成したライブラリを使います。このように、Win32 APIやDirectXに関する詳細な処理はライブラリにまとめておくとよいでしょう。こういったライブラリを作っておけば、

**Fig. 4-1** 自機を動かすプログラムの構造

ゲームの本質的な部分だけに集中して制作を進めることができます。

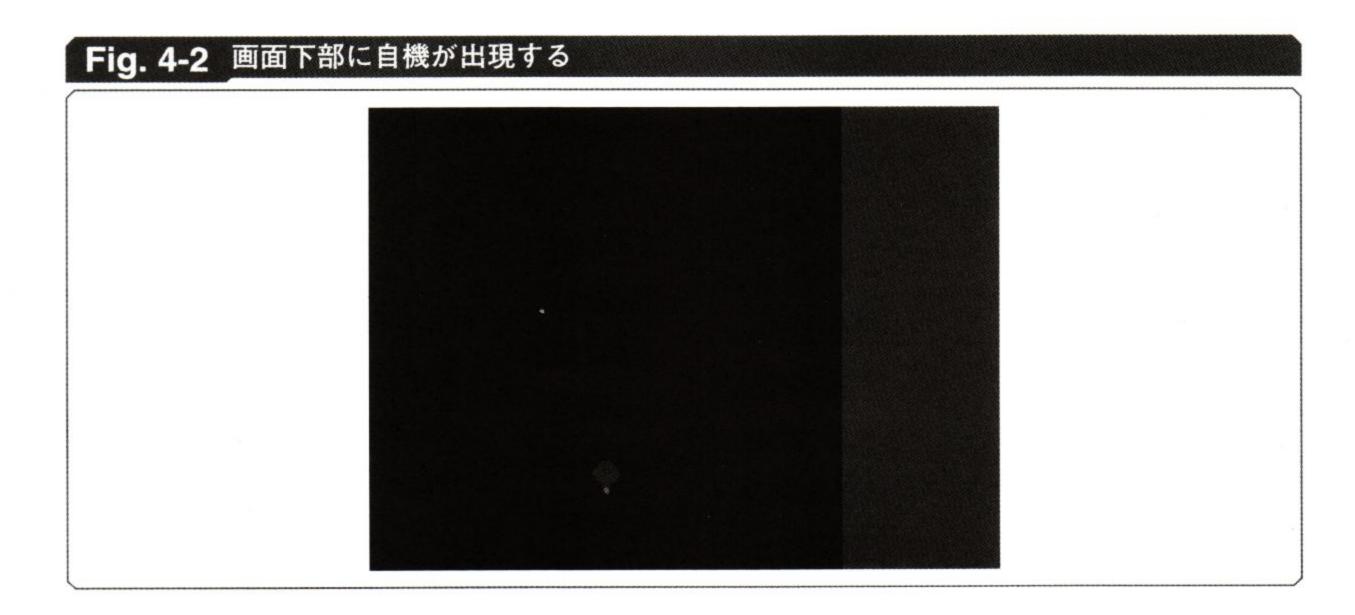
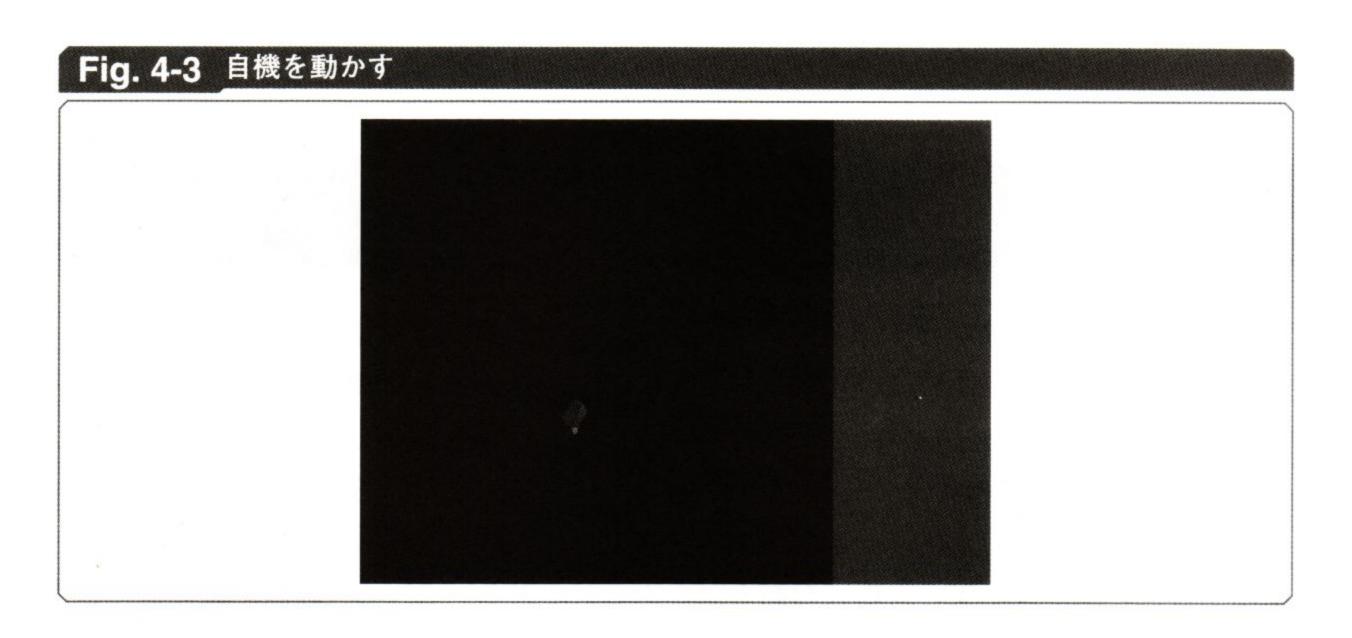
## 自機を動かすプログラム

Fig. 4-2は、自機を動かすサンプルプログラムの実行画面です。本章のサンプルのプロジェクトファイルは、付録CD-ROMの「ShtGame_MyShip1」フォルダに収録しました。実行ファイルは「ShtGame_MyShip1¥Release¥ShtGame.exe」です。

画面下部に自機(折り詰めの寿司などに入っている醤油ビンです)が出現します。カーソルキーの上下左右、またはジョイスティックの上下左右を入力すると、自機を8方向に操作することができます。自機は移動方向に応じて、左右にロール(機体を傾けること)します(Fig. 4-3)。

まだ本当に自機を動かせるだけで、ショットやビームすら発射することができません。しかし、こういったごく簡単なプログラムをシューティングゲーム制作の出発点にするとよいでしょう。最初は3Dモデルを表示したりジョイスティックの入力を読み取ったりするだけでも大変なので、このくらい単純なものから始めるのがお勧めです。

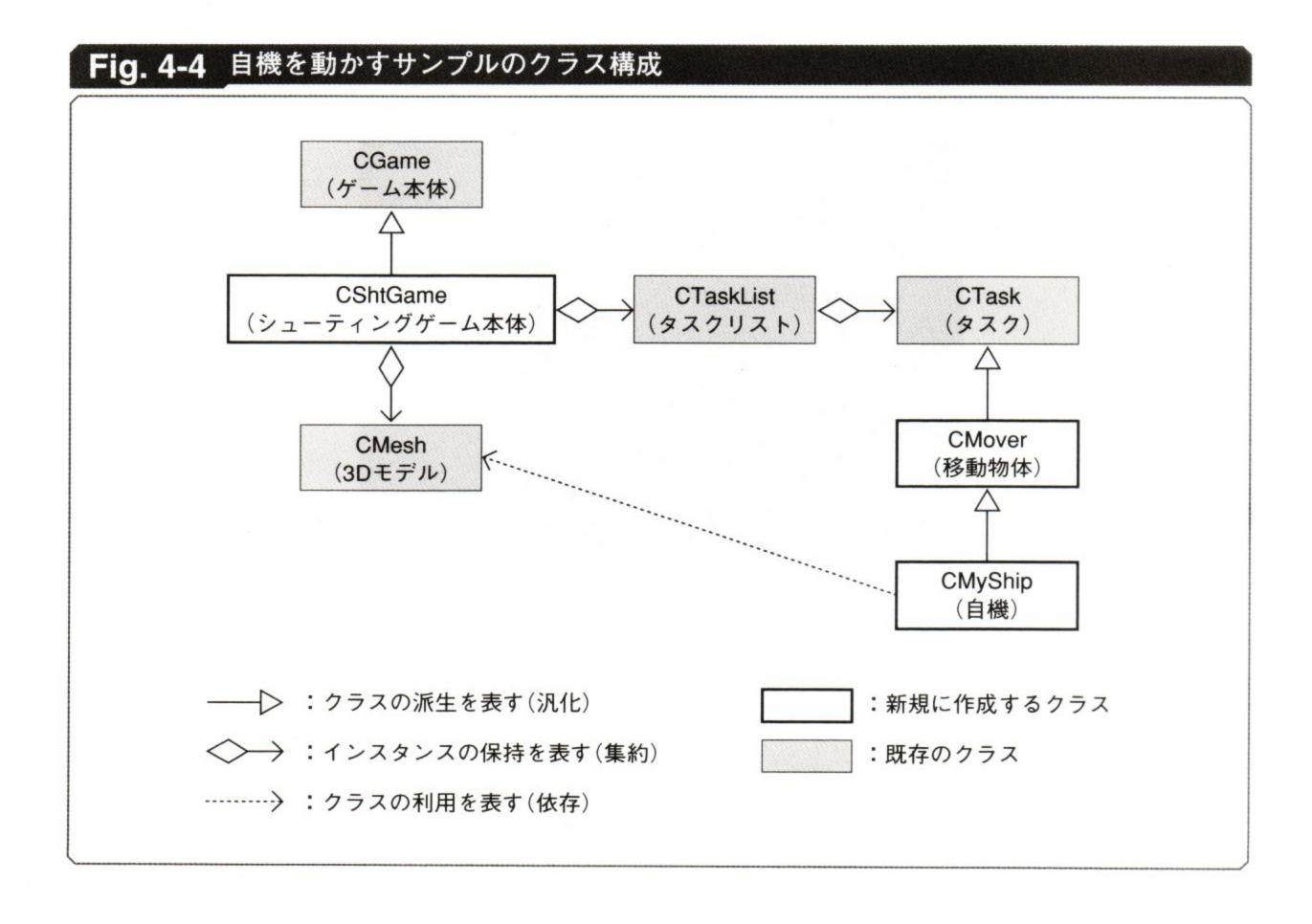
**Fig. 4-2** 画面下部に自機が出現する

**Fig. 4-3** 自機を動かす

## クラス構成

初めに、自機を動かすサンプルのクラス構成を紹介します(Fig. 4-4)。ここでは、UML (Unified Modeling Language)の記法を用いました。矩形はクラスを表し、矢印はクラス間の関係を表します。白抜きの三角形がついた矢印はクラスの派生を、白抜きの菱形がつい

**Fig. 4-4** 自機を動かすサンプルのクラス構成

CGame
(ゲーム本体)
CShtGame
(シューティングゲーム本体)
CTaskList
(タスクリスト)
CTask
(タスク)
CMesh
(3Dモデル)
CMover
(移動物体)
CMyShip
(自機)

：クラスの派生を表す(汎化)
：インスタンスの保持を表す(集約)
：クラスの利用を表す(依存)
：新規に作成するクラス
：既存のクラス

た矢印はインスタンスの保持を、破線の矢印はクラスの利用を表します。

自機を動かすだけの単純なプログラムながらも、それなりに多くのクラスを使います。しかし、ほとんどはChapter 2とChapter 3で作成ずみのクラスです。Fig. 4-4で灰色で示したクラスは作成ずみのクラス、太枠のクラスが新規に作成するクラスです。

各クラスの役割は以下のとおりです。

### CGameクラス

Chapter 2で作成したゲーム本体のクラスです（P. 20）。ウィンドウの生成と管理、ゲーム進行の制御などを行います。

### CShtGameクラス

本章で作成するシューティングゲーム本体のクラスです。CGameクラスから派生します。また、CTaskListクラスとCMeshクラスのインスタンスを保持します。

### CMeshクラス

Chapter 2で作成した3Dモデルのクラスです（P. 30）。今回は自機の3Dモデルを描画するために使います。

### CTaskListクラス

Chapter 3で作成したタスクリストのクラスです（P. 90）。今回は自機のタスクを管理するために使います。

### CTaskクラス

Chapter 3で作成したタスクのクラスです（P. 88）。今回はこのクラスから、移動物体や自機のクラスを派生させます。

### CMoverクラス

本章で作成する移動物体のクラスです。CTaskクラスから直接に自機のクラスを派生してもよいのですが、今後、ショット・弾・敵などを作成するのに備えて、このクラスを用意しました。さまざまな移動物体に共通する処理をまとめます。

### CMyShipクラス

本章で作成する自機のクラスです。自機の移動と描画を行います。

## まずはメインルーチンから

どのようなプログラムにもメインルーチンが欠かせません。アプリケーションが起動した直後に、まず実行されるのがメインルーチンです。一般的なWindowsアプリケーションでは、WinMain関数がメインルーチンになります。

ゲームプログラムの場合、メインルーチンでは、

- ゲームの初期化
- 実行
- 進行管理
- 終了

などの処理を行います。

Windows上でゲームプログラムを作る場合には、ウィンドウの初期化やWindowsのメッセージ処理なども必要です。

単純なプログラムの場合には、メインルーチンのなかにゲームの詳細な処理を直接書いてもかまいません。ある程度複雑なプログラムの場合には、詳細な処理については別の関数か、あるいはクラスにまとめるとよいでしょう。メインルーチンではそれらの関数を呼び出すか、あるいはクラスのインスタンスを作成して利用すると、プログラムの見通しがよくなります。

List 4-1は、メインルーチンの例です。この例では、ゲーム本体の詳細な処理をCShtGameクラスにまとめました。一方、自機の移動や描画の処理はCMyShipクラスにまとめました。メインルーチンではCShtGameやCMyShipのインスタンスを作成して利用します。

**List 4-1　メインルーチン（Main.cpp）**

```
// ゲーム本体
CShtGame* Game;

// メインルーチン
INT WINAPI WinMain(HINSTANCE hinst, HINSTANCE, LPSTR, INT) {

    // ゲーム本体の生成
    // CShtGameクラスはChapter 2のCGameクラス（P.20）の派生クラス
    Game=new CShtGame();
    Game->FPS=60;
    Game->PauseInTheBackground=true;
```

```
    Game->DropFrames=false;

    // 自機の生成
    new CMyShip(0, 30);

    // ゲームの実行
    // Run関数はWindowsのメッセージ処理とゲームの進行管理を行う
    Game->Run();

    // ゲームが終了するとRun関数から戻るので、
    // CShtGameインスタンスを削除してプログラムを終了する
    delete Game;
    return 0;
}
```

## ゲーム本体に必要な機能

ゲーム本体には、

- ゲームに使うデータを保持する機能
- ゲームを進行する機能
- ゲーム画面の描画を行う機能

などが必要です。

これらの機能を、構造体・関数・クラスなどを使ってまとめます。例えば、3Dモデルをファイルからロードする関数、Direct3Dを設定する関数、ゲームの動作を行う関数、ゲーム画面の描画を行う関数などを用意します。

自機や弾のようなキャラクターも管理する必要があります。こういった管理には、Chapter 3で紹介したようなタスクシステムが使えます。タスクを使うときには、自機や弾といった種類別にタスクを分類し、それぞれ別々のタスクリストで管理すると、当たり判定処理や重ね合わせ処理などの効率化が図れます。ゲームに登場する自機は1機だけのこともありますが、これも独立した専用のタスクリストで管理します。

また、タスクリスト単位で、全タスクを動かしたり描画したりする関数を用意しておくとよいでしょう。これは、例えば、タスクリスト上にあるすべての弾（のタスク）を動かしたいとか、自機だけを描画したいといった場合に便利です。

List 4-2は、ゲーム本体の機能をまとめたクラスの例です。3DモデルにはChapter 2のメ

ッシュクラス（CMesh）を使います（P. 30）。自機は醤油（soy sauce）なので、変数名は「MeshSauce」にしました。自機の管理にはChapter 3のCTaskListクラスを使っています（P. 90）。

**List 4-2　ゲーム本体のクラス（Main.h）**

```
class CShtGame : public CGame {
protected:

    // 3Dモデルのロード
    CMesh* NewMesh(string file);

public:

    // コンストラクタ、デストラクタ
    CShtGame();
    ~CShtGame();

    // デバイスの初期化、移動、描画
    virtual void OnResetDevice();
    virtual void Move();
    virtual void Draw();

    // 3Dモデル
    CMesh *MeshSauce;

    // 自機のタスクを管理するタスクリスト
    CTaskList *MyShipList;

    // タスクを動かす関数
    void MoveTask(CTaskList* list);

    // タスクを描画する関数
    void DrawTask(CTaskList* list);
};
```

## ゲームの初期化と必要なデータのロード

ゲームプログラムが最初に行う仕事は、ゲームに必要な3Dモデルなどのデータをメモリ上にロードすることです。また、例えばタスクリストなどのクラスを使う場合には、インスタンスを生成しておきます。

List 4-3は、ゲームの初期化とデータのロードに関するプログラムの例です。

**List 4-3 初期化とデータのロード（Main.cpp）**

```
// コンストラクタ
CShtGame::CShtGame()
:   CGame("紫雨 (MURASAME)", true, false, true)
{
    // ヘルプ、終了、カーソルに関する設定
    HelpURL=ExePath+"..¥¥Help¥¥index.html";
    ConfirmExit=false;
    ShowCursor(FALSE);

    // 3Dモデルの初期化
    // ファイルから3Dモデルのメッシュ（形状データ）を読み込む
    LPDIRECT3DDEVICE9 device=Graphics->GetDevice();
    MeshSauce=NewMesh("sauce");

    // タスクリストの初期化
    MyShipList=new CTaskList(sizeof(CMyShip), 10);
}

// 3Dモデルのロード
// Chapter 2のCMeshクラスを用いる
CMesh* CShtGame::NewMesh(string file) {
    string path=ExePath+"..¥¥Model¥¥";
    CMesh* p=new CMesh(Graphics->GetDevice());
    p->LoadFromFile(path+file+".x");
    return p;
}
```

## グラフィックの初期化

次はグラフィックの初期化です。Direct3Dを使うとなると、どうしてもある程度は複雑な初期化処理が必要になってしまいます。とはいえ、こういった初期化処理はいろいろなアプリケーションに共通するものなので、一度プログラムを書いてしまえば、その後はコピーして使い回すことができます。

また、Direct3Dでは画面モードや解像度を切り替えたときなどにデバイスロストが起こります（P. 27）。このときレンダリングや行列の設定が失われてしまうため、設定を復帰させる処理が必要です。

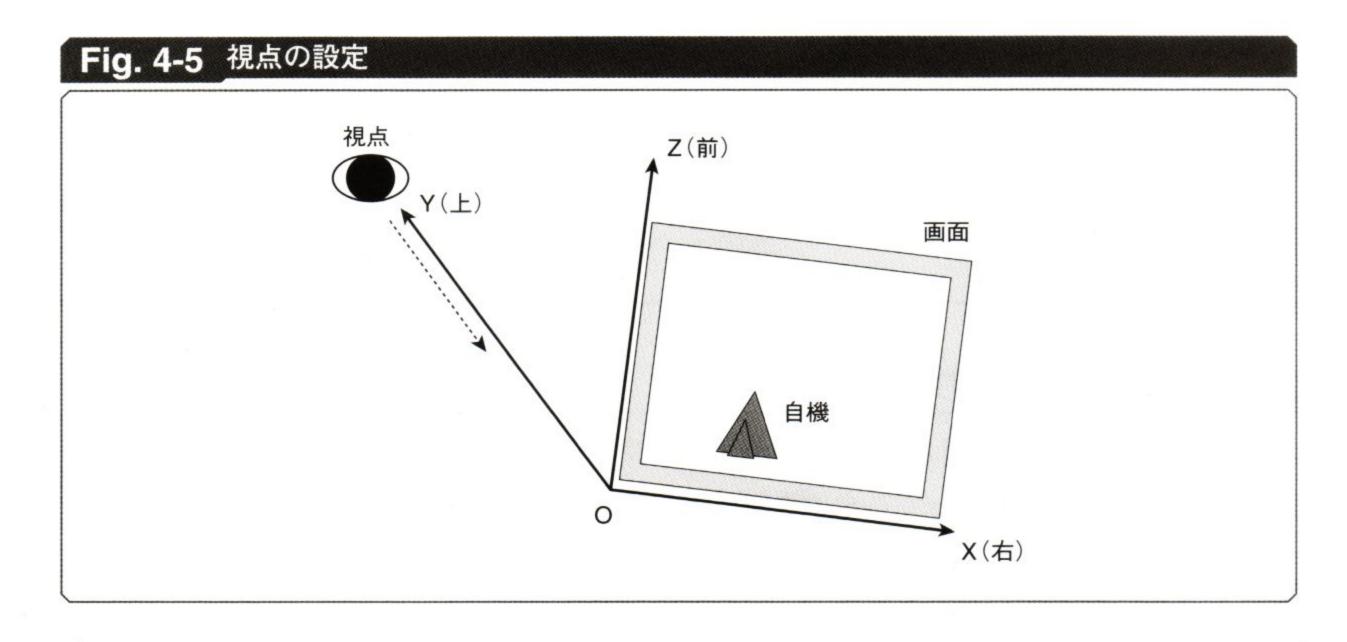
設定を復帰させる行列には、ビュー行列や射影行列などがあります。ビュー行列は3D空間内の座標系（ワールド座標系）を視点から見た座標系（ビュー座標系）に変換します。射影行列はビュー座標系を画面上の座標系に変換します。

3Dグラフィックを利用して2Dシューティングゲームの画面を描画するには、例えばFig. 4-5のように視点を設定します。ここでは左手座標系と平行投影を使用し、視点はY軸正方向から原点を見る方向にしています。自機などの3Dモデルは、X軸正方向が右、Y軸正方向が上、Z軸正方向が前となるようにデザインします。

なお、左手座標系というのは、左手の親指・人さし指・中指がそれぞれ90度の角度をなすように開いたとき、親指がX軸、人さし指がY軸、中指がZ軸となるような座標系です。右手座標系では、逆に右手の親指・人さし指・中指を開いたときの方向に座標軸を取ります。

また、3Dモデルを照らす光源の設定も必要です。いちばん簡単な例としては、斜め上方向から照らす平行光源を1個設定する方法があります。3Dモデルに陰影をつけて表示する場合には、少なくとも1個は光源を設定しておく必要があります。

List 4-4は、グラフィックの初期化を行うプログラムです。このOnResetDevice関数はChapter 2のゲームクラス（CGame）の関数で、Direct3Dデバイスがリセットされたときに、デバイスの状態を復帰するために呼び出されます。

**Fig. 4-5 視点の設定**

**List 4-4 グラフィックの初期化（Main.cpp）**

```
void CShtGame::OnResetDevice() {
    LPDIRECT3DDEVICE9 device=Graphics->GetDevice();

    // アルファブレンディングの設定
    device->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
    device->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
    device->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
```

```
    // レンダリングの設定
    // ライティングやZバッファに関する設定を行う
    device->SetRenderState(D3DRS_DITHERENABLE, FALSE);
    device->SetRenderState(D3DRS_SPECULARENABLE, FALSE);
    device->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
    device->SetRenderState(D3DRS_LIGHTING, TRUE);
    device->SetRenderState(D3DRS_ZENABLE, TRUE);
    device->SetRenderState(D3DRS_AMBIENT, 0x006F6F6F);
    device->SetRenderState(D3DRS_ALPHAREF, (DWORD)0x00000001);
    device->SetRenderState(D3DRS_ALPHATESTENABLE, TRUE);
    device->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);

    // テクスチャ処理方法の設定：
    device->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);
    device->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
    device->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
    device->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_MODULATE);
    device->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
    device->SetTextureStageState(0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE);

    // ワールド行列
    D3DXMATRIX mat_world;
    D3DXMatrixIdentity(&mat_world);
    device->SetTransform(D3DTS_WORLD, &mat_world);

    // ビュー行列
    D3DXMATRIX mat_view;
    D3DXVECTOR3 vec_from=D3DXVECTOR3(0, 100, 0);
    D3DXVECTOR3 vec_lookat=D3DXVECTOR3(0, 0, 0);
    D3DXVECTOR3 vec_up=D3DXVECTOR3(0, 0, 1);
    D3DXMatrixLookAtLH(&mat_view, &vec_from, &vec_lookat, &vec_up);
    device->SetTransform(D3DTS_VIEW, &mat_view);

    // 射影行列
    D3DXMATRIX mat_proj;
    D3DXMatrixOrthoLH(&mat_proj, 100, 100, 1, 1000);
    device->SetTransform(D3DTS_PROJECTION, &mat_proj);

    // ライティング
    Graphics->SetLight(
        0, D3DLIGHT_DIRECTIONAL, -1.0f, -1.0f, -1.0f, 1000);
}
```

# 移動する物体に共通する機能

いよいよ自機を動かす準備に入ります。まずは、自機・ショット・弾・敵といったさまざまな移動物体（キャラクター）に共通する処理をまとめてみましょう。

移動物体には座標が不可欠です。画面上の位置はX座標とY座標で表します。重ね合わせの上下を表す場合には、それに加えてZ座標も使うことができます。

ゲームで座標系を扱う方法はいろいろありますが、例えば画面右側をX軸正方向、画面下側をY軸正方向とする方法があります（Fig. 4-6）。

画面のサイズについても、自由に決めることができます。これは、キャラクターを描画するときなどに、ゲームで扱う座標を画面上の座標に変換すればよいからです。ここではサイズを100×100としました。

3Dモデルを表示する際には、ゲームの座標を3Dグラフィックの座標に変換します。グラフィックの初期化のところで述べたように、3D座標はちょうど画面を上から見下ろすように設定してあります（Fig. 4-7）。そのため、例えば自機のゲーム座標が（x, y）ならば、3D座標は（x, 0, -y）になります。

3D座標をそのままゲーム座標として使うことも不可能ではないのですが、2DゲームではFig. 4-6のようにX座標とY座標だけを考えた方がわかりやすくなります。そのため、このように座標を使い分けるのも1つの方法です。

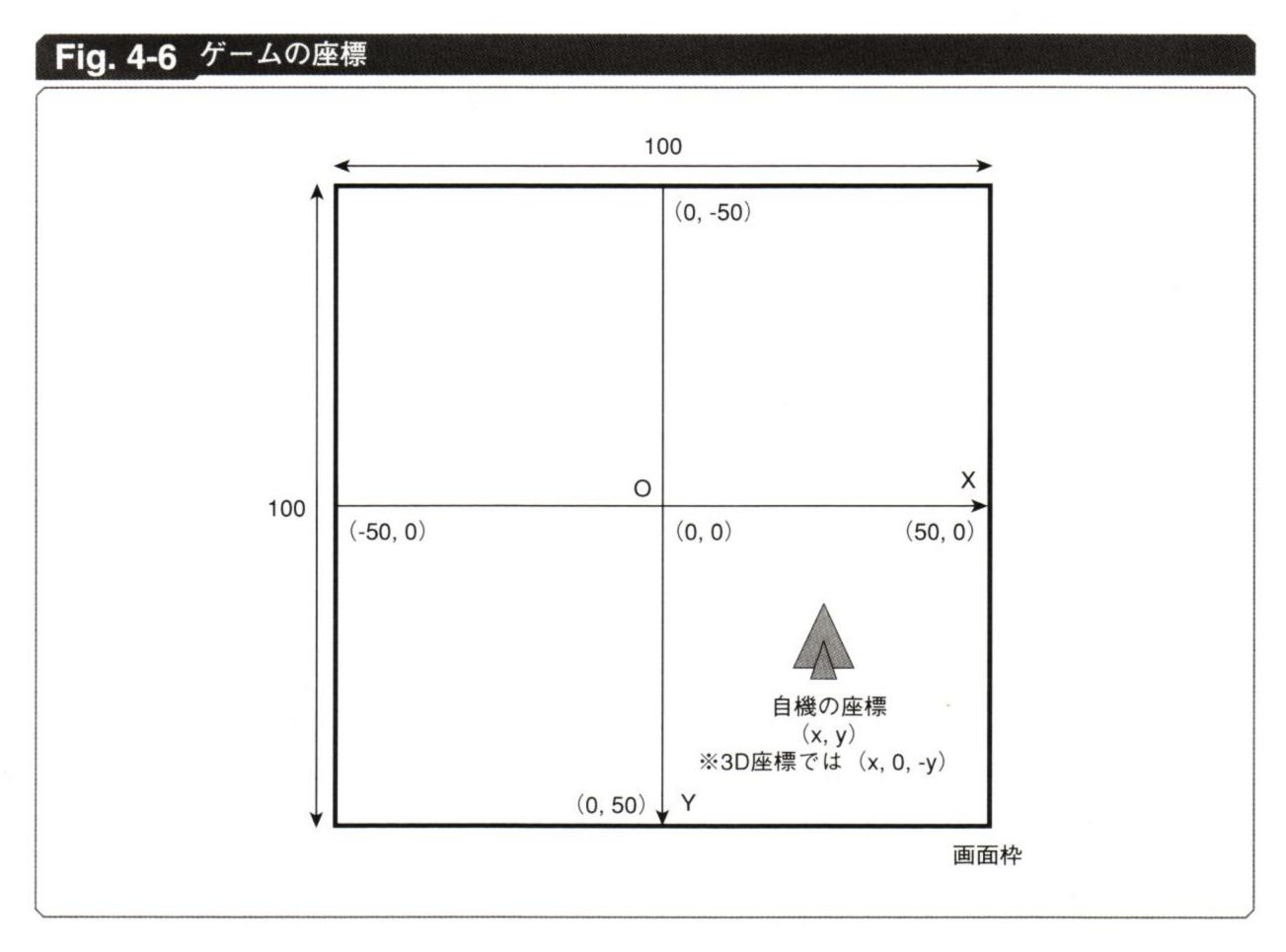
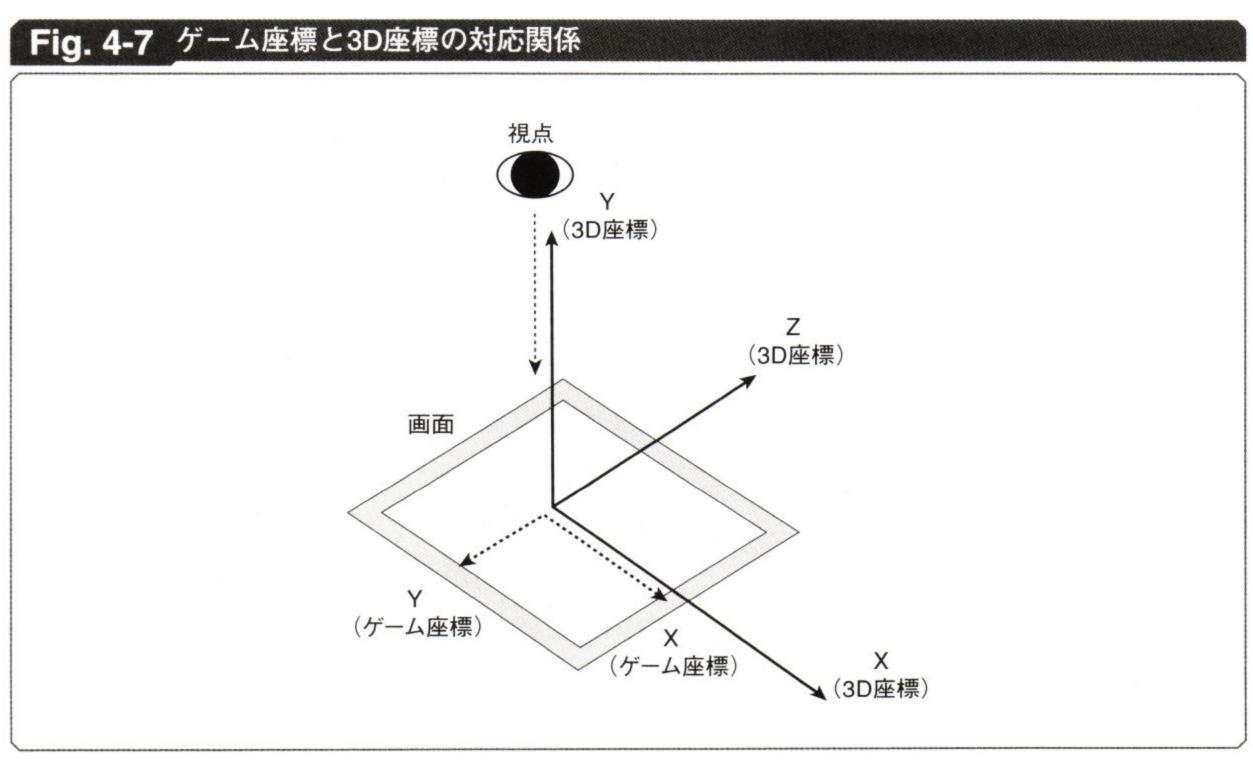
移動物体には移動と描画の機能が必要です。ただし、移動と描画の詳細な処理内容については、自機や弾といった物体の種類によって異なります。

そこで、もしクラスを利用する場合には、移動物体のクラスでは移動と描画を行う関数を仮想関数として定義しておきます。そして、移動物体クラスから派生する自機クラスや弾クラスでこれらの仮想関数をオーバーライドして、それぞれの物体にふさわしい処理を記述します。

List 4-5は、移動物体に関するクラスです。弾が画面外に出てしまったり、敵が爆発した場合には、そのタスクを消去しなければなりません。消去すべきタスクを示すために、この例では移動を行うMove関数の戻り値を利用しています。

**Fig. 4-6 ゲームの座標**

100
(0, -50)
O
X
100
(-50, 0)
(0, 0)
(50, 0)
自機の座標
(x, y)
※3D座標では (x, 0, -y)
(0, 50)
Y
画面枠

**Fig. 4-7 ゲーム座標と3D座標の対応関係**

**List 4-5 移動物体のクラス（Mover.h）**

```
class CMover : public CTask {
public:

	// 座標
	// 他のクラスから操作する都合上、publicメンバにした
	// privateメンバにしてアクセス用のメソッドを用意してもよい
	float X, Y, Z;

	// コンストラクタ
	// 引数で受け取った座標をメンバ変数に設定する
	CMover(CTaskList* task_list, float x, float y, float z)
	:	CTask(task_list), X(x), Y(y), Z(z)
	{}

	// 移動
	// CMoverクラスの派生クラスでは
	// この仮想関数をオーバーライドする
	// 移動後に移動物体を消去する場合にはfalseを返し
	// 消去しない場合にはtrueを返す
	virtual bool Move() { return true; }

	// 描画
	// CMoverクラスの派生クラスでは
	// この仮想関数をオーバーライドする
	virtual void Draw() {}
};
```

## 自機の移動

それでは、実際に自機の移動処理を作成しましょう。まず、フレーム（1/60秒などの一定時間間隔）ごとにプレイヤーの入力を読み取ります。そして、入力内容に応じて少しずつ自機を動かします。

自機の移動に関しては、次のようなポイントがあります。

### 入力の読み取りと移動

入力内容に応じて自機の速度を設定します。例えば左が入力された場合には、自機が左向きの速度を持つように、速度のX成分を負の値に設定します。また、例えば右上が入力された場合には、速度のX成分を正の値に、速度のY成分を負の値に設定します。

### ロール

左右に移動したときに自機を傾ける処理です（Fig. 4-8）。ロールを行うと、自機が空間を飛んでいる雰囲気を出すことができます。キーボードやジョイスティックの左右への入力があったら、機体を傾けます。左右への入力がない場合には、傾いた自機を少しずつ水平な状態に戻します。

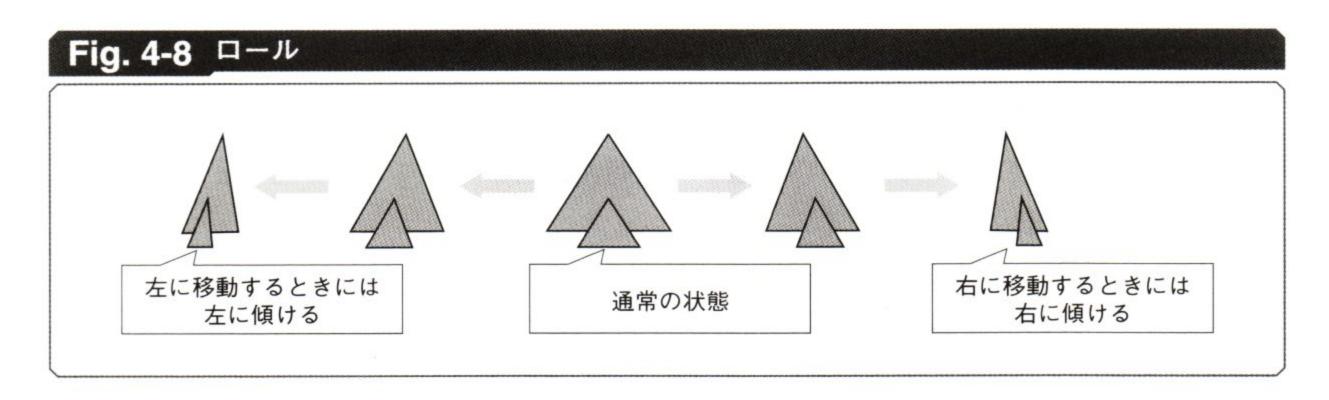
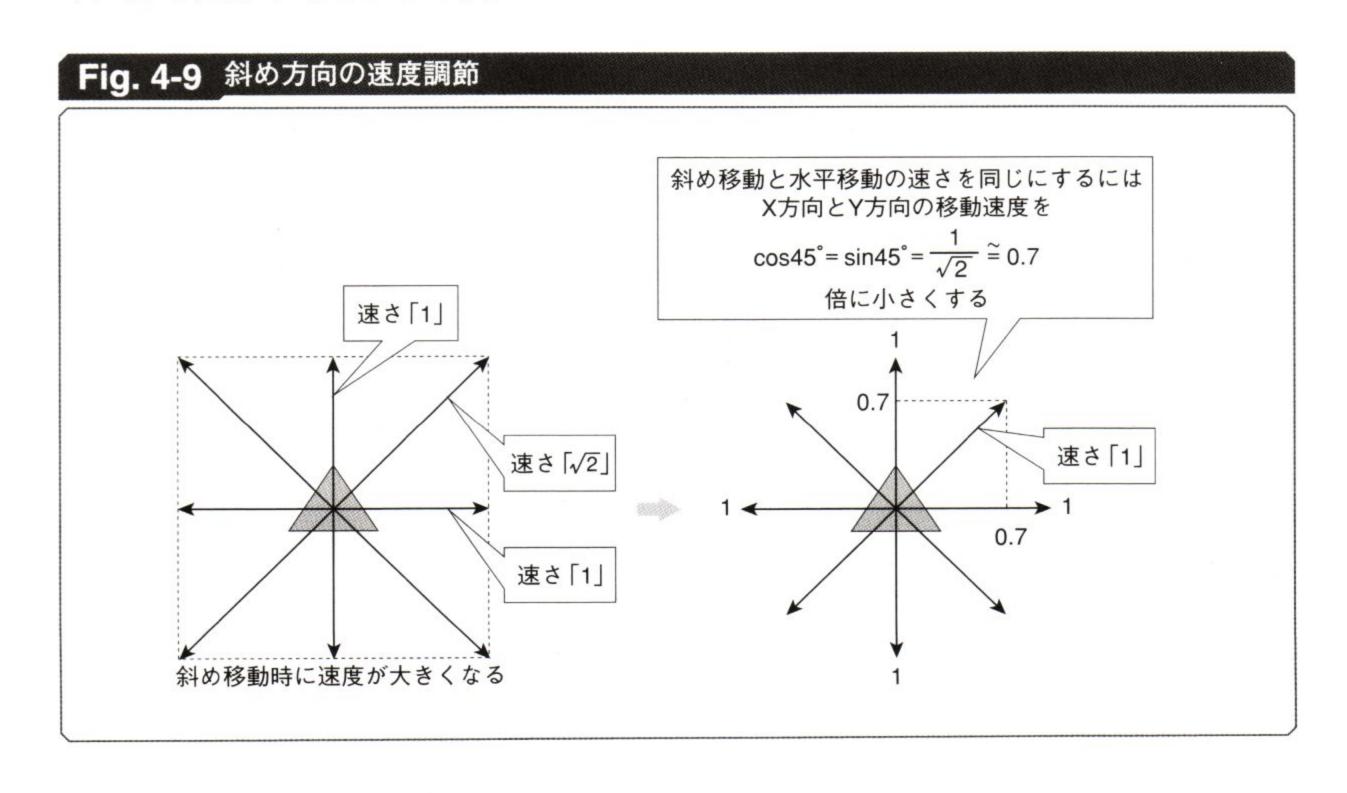
**Fig. 4-8 ロール**

### 斜め方向の速度調節

Fig. 4-6のような座標を利用する場合には、斜め移動と水平移動の速度の違いに注意が必要です。例えば、自機の速度を1にしたい場合、右へ進むには速度のX成分を1にします。同様に、下に進むときにはY成分を1にします。しかし、右下へ斜めに進むときに、X成分を1、Y成分を1としてしまうと、思ったような結果になりません。自機の速度が1ではなく、$\sqrt{2}$（約1.4）になるからです。

**Fig. 4-9 斜め方向の速度調節**

斜め移動と水平移動の速さを同じにするには、斜め移動時にX方向とY方向の移動速度を約0.7倍に小さくする必要があります（Fig. 4-9）。約0.7倍というのは、$\sqrt{2}$の逆数（1/sqrt(2.0f)）です。

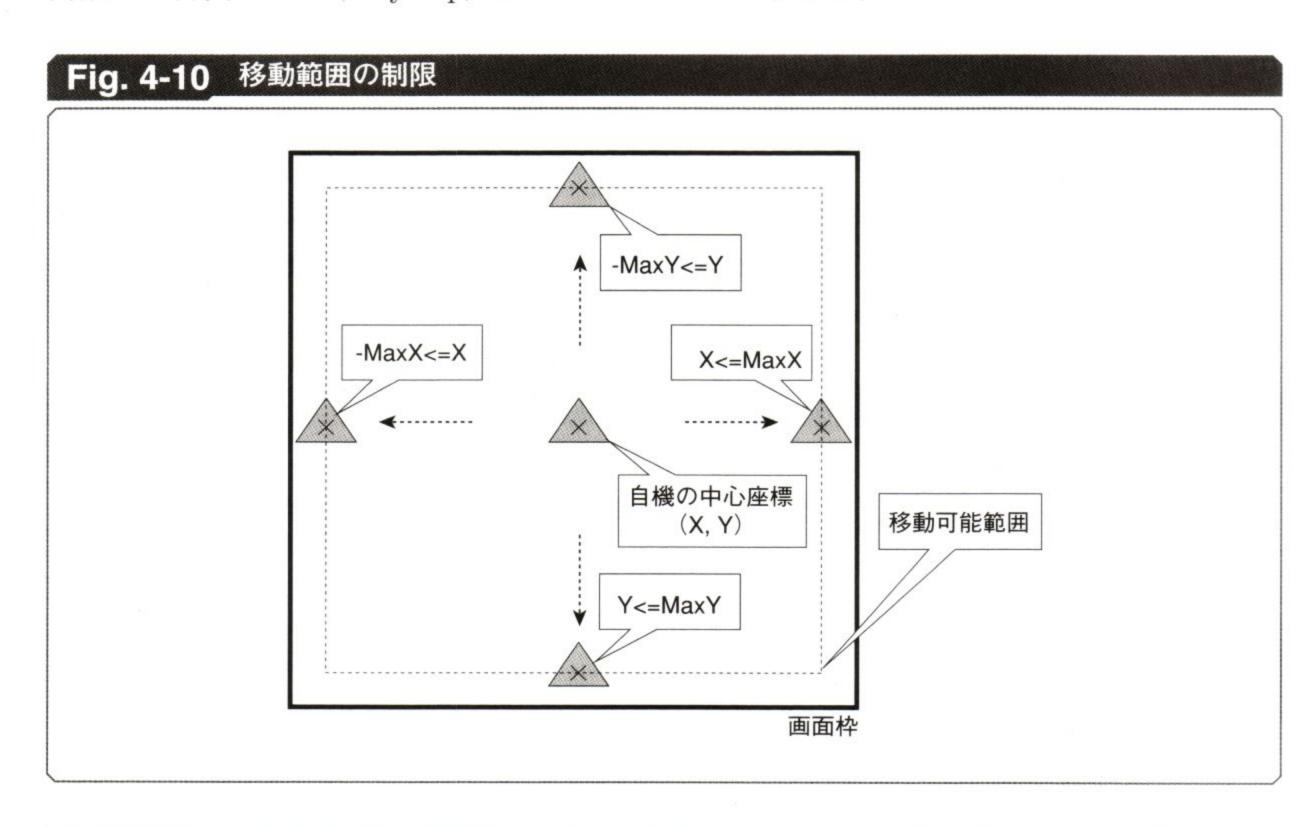
速度調整がない場合、斜めに移動すると急に自機が速くなってしまいます。速度調節を行うかどうかはゲームによって違いますが、本書では調整することにしました。

## 移動範囲の制限

自機が画面枠からはみ出ないように、移動範囲を制限します（Fig. 4-10）。例えば、自機の座標を更新した後に、自機の中心座標が一定範囲以内に入っているかどうかを調べ、出ていたら範囲内に戻します。

※

List 4-6は、自機の移動をまとめたプログラムです。移動物体クラス（CMover）のMove関数を、自機クラス（CMyShip）でオーバーライドしました。

**Fig. 4-10 移動範囲の制限**

**List 4-6 自機の移動（MyShip.cpp）**

```
bool CMyShip::Move() {

    // 移動範囲
    static const float MaxX=46, MaxY=46;
```

```
    // ロール速度と範囲
    static const VRoll=0.02f, MaxRoll=0.1f;

    // 斜め移動の速度調整用定数
    static const sqrt2=1/sqrt(2.0f);

    // 速度
    // この値を変更すると自機の移動速度が変わる
    float vx=0.8f, vy=0.8f;

    // キーボードとジョイスティックの入力を受け取る
    // Chapter 2で作成したInputクラスを使用（P. 37）
    const CInputState* is=Game->GetInput()->GetState(0);

    // 左右移動とロール
    if (is->Left) {
        vx=-vx;
        if (Roll>-MaxRoll) Roll-=VRoll;
    } else
    if (is->Right) {
        if (Roll<MaxRoll) Roll+=VRoll;
    } else {
        vx=0;
        Roll+=(Roll<0)?VRoll:-VRoll;
    }

    // 上下移動
    if (is->Up) vy=-vy; else if (!is->Down) vy=0;

    // 斜め方向の速度調節
    if (vx!=0 && vy!=0) {
        vx*=sqrt2;
        vy*=sqrt2;
    }

    // 移動
    X+=vx;
    Y+=vy;

    // 移動範囲の制限
    if (X<-MaxX) X=-MaxX; else if (MaxX<X) X=MaxX;
    if (Y<-MaxY) Y=-MaxY; else if (MaxY<Y) Y=MaxY;

    // このプログラムでは自機を消去することはないので
    // 常にtrueを返す
    return true;
}
```

### 自機の描画

移動させた後で新しい位置に自機を描画することによって、自機が実際に画面上で動いたように見せることができます。自機を描画する方法はいろいろあります。例えば、3Dグラフィックを使う場合には、自機の座標に自機の3Dモデルを描画します。

ゲームの座標系と3Dの座標系が異なる場合には、座標の変換が必要です。本書の場合、ゲーム座標系の座標（X, Y, Z）は、3D座標系の座標（X, Z, -Y）に対応します（P. 116）。

機体をロールさせるには、3Dモデルの回転も必要です。また、3Dモデルの立体感を出すためには、少し傾けて表示すると効果的です。2Dグラフィックのゲームでも、キャラクターが真上からの視点ではなく、斜め上からの視点で描かれているのをよく見かけます。

List 4-7は、自機の描画に関するプログラムです。

**List 4.7　自機の描画（MyShip.h、MyShip.cpp）**

```
void CMyShip::Draw() {

	// 描画色の設定
	// ここでは、もともとの3Dモデルの色を使用
	Game->MeshSauce->SetColor(
		D3DXCOLOR(1, 1, 1, 1), D3DXCOLOR(0, 0, 0, 0));

	// 3Dモデルの描画
	// Chapter 2のCMeshクラスを使用する（P. 30）
	// 第3引数は回転角度、第4引数は回転順序を指定する
	Game->MeshSauce->Draw(
		D3DXVECTOR3(X, Z, -Y), D3DXVECTOR3(1, 1, 1),
		D3DXVECTOR3(0.125f, 0, -Roll), TO_ZYX);
}
```

## 自機の機能をまとめる

自機の機能は自機クラス（CMyShip）にまとめることにします。自機クラスは移動物体クラス（CMover）から派生し、移動物体クラスはタスククラス（CTask）から派生しています。

タスククラスはChapter 3で作成しました（P. 88）。このクラスを使う際には、new演算子とdelete演算子をオーバーロードして、operator_new関数とoperator_delete関数にタスクリストを渡す必要があります。ここでは自機を管理するためのタスクリスト

(MyShipList) を指定します。

List 4-8は、自機の機能をまとめたクラスです。自機クラス (CMyShip) は移動物体クラス (CMover) から派生するので、自機クラスのコンストラクタでは、移動物体クラスのコンストラクタを呼び出します。

**List 4-8 自機のクラス (MyShip.h、MyShip.cpp)**

```
class CMyShip : public CMover {
protected:

    // ロールの角度
    float Roll;

public:

    // new演算子、delete演算子
    void* operator new(size_t t) {
        return operator_new(t, Game->MyShipList);
    }
    void operator delete(void* p) {
        operator_delete(p, Game->MyShipList);
    }

    // コンストラクタ
    CMyShip::CMyShip(float x, float y)
    :   CMover(Game->MyShipList, x, y, MYSHIP_Z), Roll(0)
    {}

    // 移動、描画
    virtual bool Move();
    virtual void Draw();
};
```

## フレームごとにゲーム全体を進行させる

フレームごとに少しずつ自機や弾などの移動処理を呼び出すことによって、ゲーム全体を進行させます。タスクシステムを使う場合には、タスクリストにあるすべてのタスクを移動 (実行) させます。

List 4-9は、ゲーム全体を進行させるプログラムです。ここではChapter 3で作成したCTaskIterクラスを使っています (P. 97)。CTaskIterクラスは、タスクリスト上にあるすべてのタスクに対して繰り返し処理を行うためのクラスです。

**List 4-9　ゲーム全体の進行（Main.cpp）**

```
// ゲーム全体の動作
// Chapter 2で作成したCGameクラスの機能により（P. 20）
// このMove関数はフレームごとに呼び出される
void CShtGame::Move() {

    // 自機のタスクを動かす
    MoveTask(MyShipList);
}

// タスクの動作
void CShtGame::MoveTask(CTaskList* list) {

    // すべてのタスクについて繰り返す
    for (CTaskIter i(list); i.HasNext(); ) {
        CMover* mover=(CMover*)i.Next();

        // タスクのMove関数を呼び出す
        // Move関数がfalseを返した場合にはタスクを消去する
        if (!mover->Move()) i.Remove();
    }
}
```

## ゲーム画面の描画

あとはゲーム画面を描画すれば、自機が動いている様子を見ることができます。通常はフレームごとに、ゲーム全体の進行処理に続いて、ゲーム画面の描画処理を行います。

ゲームによっては、画面をゲーム用の領域とスコアなどを表示するための領域に分けています。特にPCのディスプレイのような横長の画面で縦スクロールのシューティングゲームを動かす場合に多いようです。本書では、画面左側に正方形のゲーム領域を確保し、残りの領域をスコア領域にしています。

タスクシステムを使う場合には、移動処理と同様に、すべてのタスクについて描画処理を行います。現段階では自機のタスクだけを描画することになります。

List 4-10は、ゲーム画面の描画を行うプログラムです。

**List 4-10　ゲーム全体の描画（Main.cpp）**

```
// 描画
// Chapter 2で作成したCGameクラスの機能により（P. 20）
// 画面の描画が必要になったときに呼び出される
void CShtGame::Draw() {
    LPDIRECT3DDEVICE9 device=Graphics->GetDevice();

    // ゲーム領域の初期化
    D3DVIEWPORT9 viewport;
    int w, h;
    w=Graphics->GetWidth();
    h=Graphics->GetHeight();
    viewport.X=0;
    viewport.Y=0;
    viewport.Width=h;
    viewport.Height=h;
    viewport.MinZ=0;
    viewport.MaxZ=1;

    // ゲーム領域を描画対象にする
    device->SetViewport(&viewport);

    // ゲーム領域を黒色でクリアする
    Graphics->Clear(ColBlack);

    // タスクの描画
    DrawTask(MyShipList);

    // スコア領域の初期化
    viewport.X=h;
    viewport.Width=w-h;

    // スコア領域を描画対象にする
    device->SetViewport(&viewport);

    // スコア領域を紫色でクリアする
    Graphics->Clear(D3DCOLOR_XRGB(100, 50, 80));

    // スコアや残機といった情報を表示する場合は
    // ここに処理を追加する（P. 227）
}

// タスクの描画
// タスクリストに属するすべてのタスクについてDraw関数を呼び出す
// ここでは自機の描画に使用する
```

```
void CShtGame::DrawTask(CTaskList* list) {
    for (CTaskIter i(list); i.HasNext(); ) {
        ((CMover*)i.Next())->Draw();
    }
}
```

### 自機を動かすプログラムのまとめ

自機を動かすプログラムについてひととおり解説しましたが、いかがでしょうか。単純な機能のわりに、けっこう複雑なプログラムだと感じられるかもしれません。その一方で、Chapter 2とChapter 3で作成したライブラリを使用したおかげで、Win32 APIやDirectXまわりの煩雑な処理はかなり隠蔽されているのではないかと思います。

いずれにしても、こういったごく単純なプログラムを作るところから、シューティングゲームの制作は始まります。本章でこれまでに解説したプログラムは、シューティングゲームの基礎となる部分です。誌面とあわせて、付録CD-ROMに収録したサンプルとそのプロジェクトを、ぜひご覧になってみてください。

また、自機の移動に関する定数を変更して、自機の移動速度や移動範囲を変えてみてください。パラメータを変更することで動きがどのように変化するのか、感覚をつかんでおくとゲームの設計がしやすいでしょう。

## ショットとビームの発射

自機を動かすことができたら、制作をもう一歩進めましょう。次に何を作るのかはお好みしだいですが、本書ではショットとビームが発射できるようにしたいと思います（Fig 4-11）。

ここからのサンプルのプロジェクトは、付録CD-ROMの「ShtGame_MyShip2」フォルダに収録しています。実行ファイルは「ShtGame_MyShip2¥Release¥ShtGame.exe」です。

ショットとビームは、1つのボタンで撃ち分けられるようにしました。ボタンはジョイスティックのボタン「0」か、キーボードの「Z」キーを使います。操作は以下のとおりです。

- ボタンをリズミカルに叩くと途切れずにショットが出る
- ボタンを押しっぱなしにするとパワーがたまり、十分にたまるとビームが出る

これはいわゆるセミオート連射の発展型です。セミオート連射では、ボタンを本気で連射しなくても、ある程度の速さで連射しているだけで、高速にショットを連射することができます。それに加えて、ボタンを押しっぱなしにしたときにビームを発射できるようにしたのが、この方式です。この方式を採用したゲームとしては、「首領蜂」や「怒首領蜂」をはじめとするケイブ（http://www.cave.co.jp/）の作品が有名です。

この方式を採用した多くの作品では、ショットよりもビームの方が攻撃力が高くなっています。また、ビームを発射している間には自機の移動速度が遅くなります。そのため、以下のようにショットとビームを使い分けるのが一般的です。

### ショット

攻撃力は弱めだが、自機の移動速度が速く、攻撃が広範囲に広がるため、小さくて軟らかいザコ敵などを一掃するために使います。

### ビーム

自機の移動速度が遅く、攻撃範囲は狭いが、攻撃力が高いため、弾幕を細かく避けながら大型のボスなどを削るために使います。

※

このようにショットとビームの特性を変えて、さらに自機の移動速度にも変化をつけることによって、攻撃を使い分ける楽しさが生まれます。ショットとビームにかぎらず、複数の攻撃手段を用意するときには、それぞれの攻撃に特徴を持たせて、使い分けが楽しくなるようなゲームデザインを心がけるとよいでしょう。

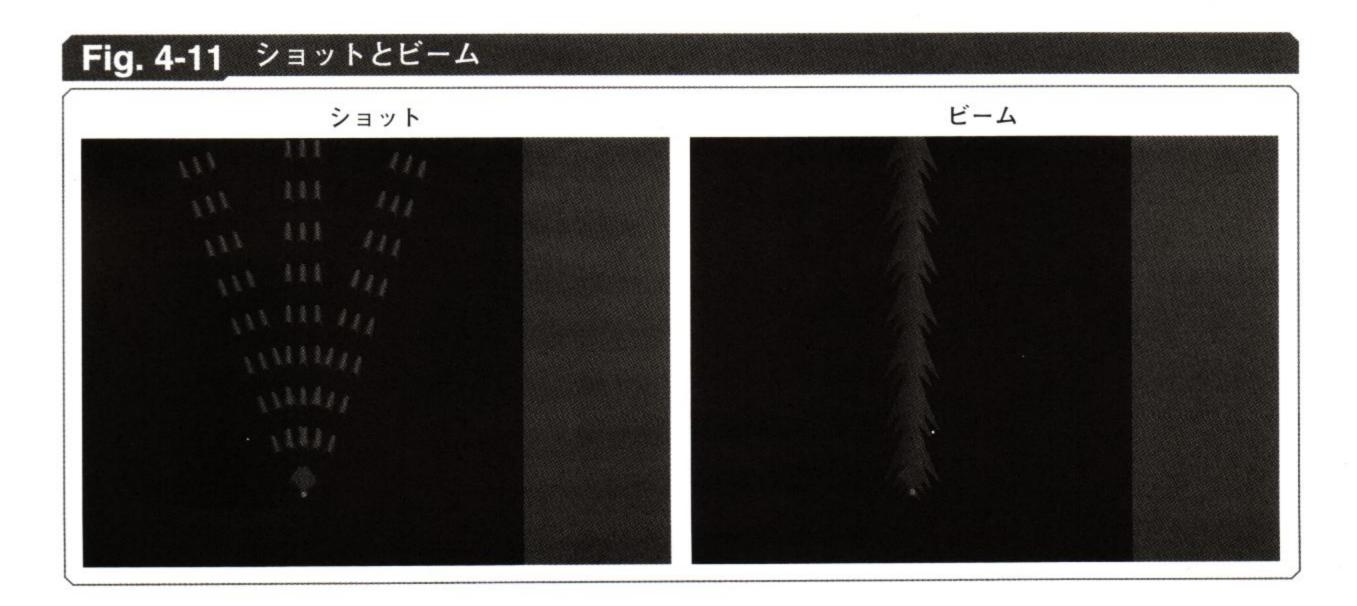



**Fig. 4-11** ショットとビーム

# ショットとビームの仕組み

ショットとビームは、Fig. 4-12〜14のように実現します。パワーとゲージという2つの変数を使って、ショットとビームの切り替えを制御します。

パワーはビームを出すかどうかを判定するために使います。ボタンを押し続けているとパワーは増加し、ボタンを離すとパワーは減少します。パワーが定数BeamMinPowerよりも大きくなると、ビームが発射されます。

ゲージはショットを連射するために使います。ボタンを押すとゲージは定数ShotMaxGaugeになり、ボタンを離すとゲージは減少します。ゲージが一定値以上（ここでは1以上）の間は、ショットが発射されます。

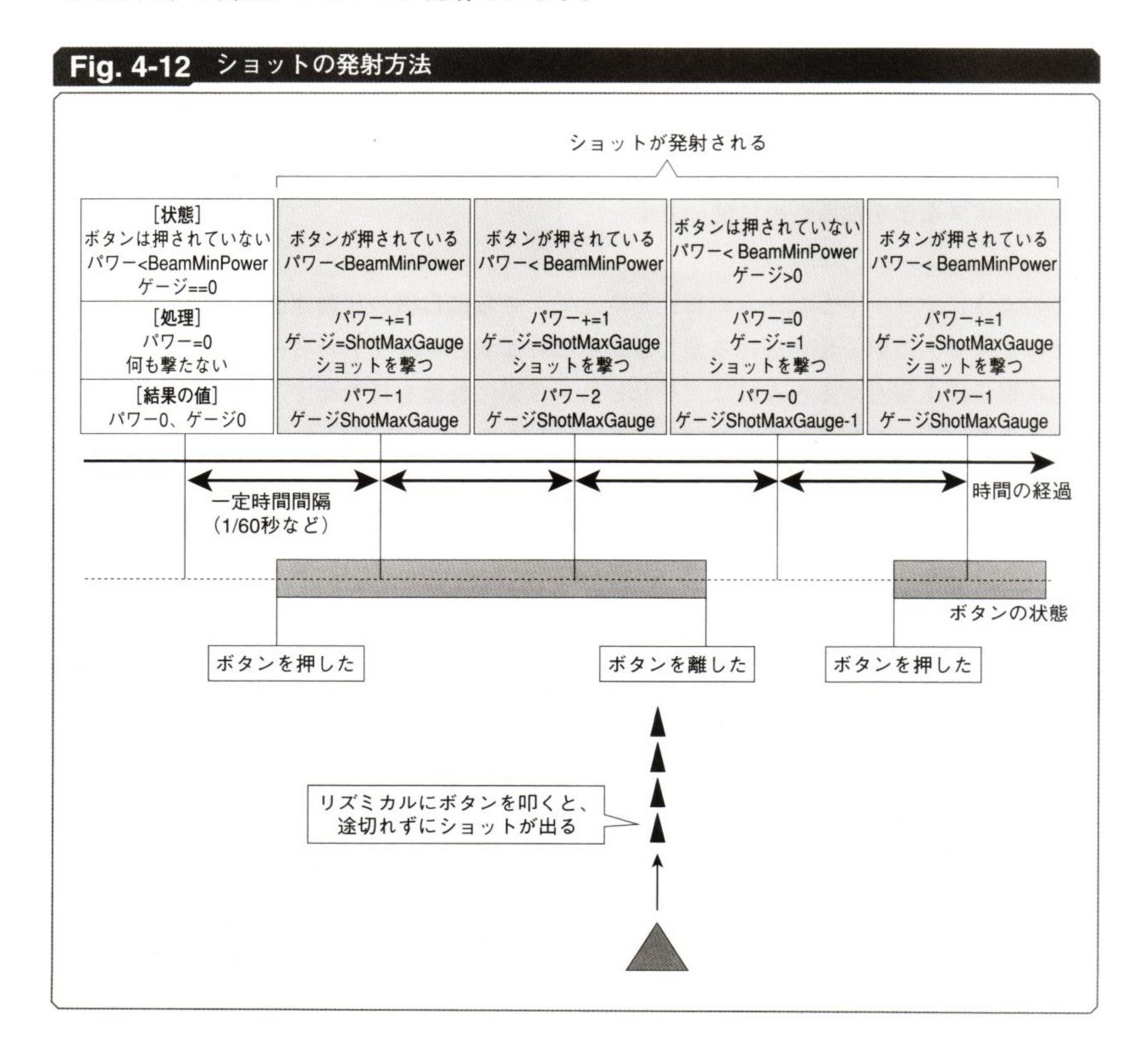
**Fig. 4-12** ショットの発射方法

ボタンを一定以上の速さで連射していると、ゲージが1以上に保たれるため、ショットが連射できます。ボタンを押しっぱなしにすると、パワーがたまるまではショットが出ますが、パワーがたまるとビームに変わります。ショット連射中にボタンを離すと、少し連射が続いてから停止します。一方、ビーム発射中にボタンを離すと、ビームは即座に停止します。

Fig. 4-15は、ショットとビームを加えたプログラムのクラス構成です。ショットとビームを実現するために、ショットクラス（CShot）とビームクラス（CBeam）を新たに作成しました。そして、既存のクラスも拡張します。

ショットクラスとビームクラスは、自機クラス（CMyShip）と同様に、移動物体クラス（CMover）の派生クラスです。これらのクラスは、ショットやビームを表示するために、Chapter 2で作成したメッシュクラス（CMesh）を利用します（P. 30）。

**Fig. 4-13 ショットの停止方法**

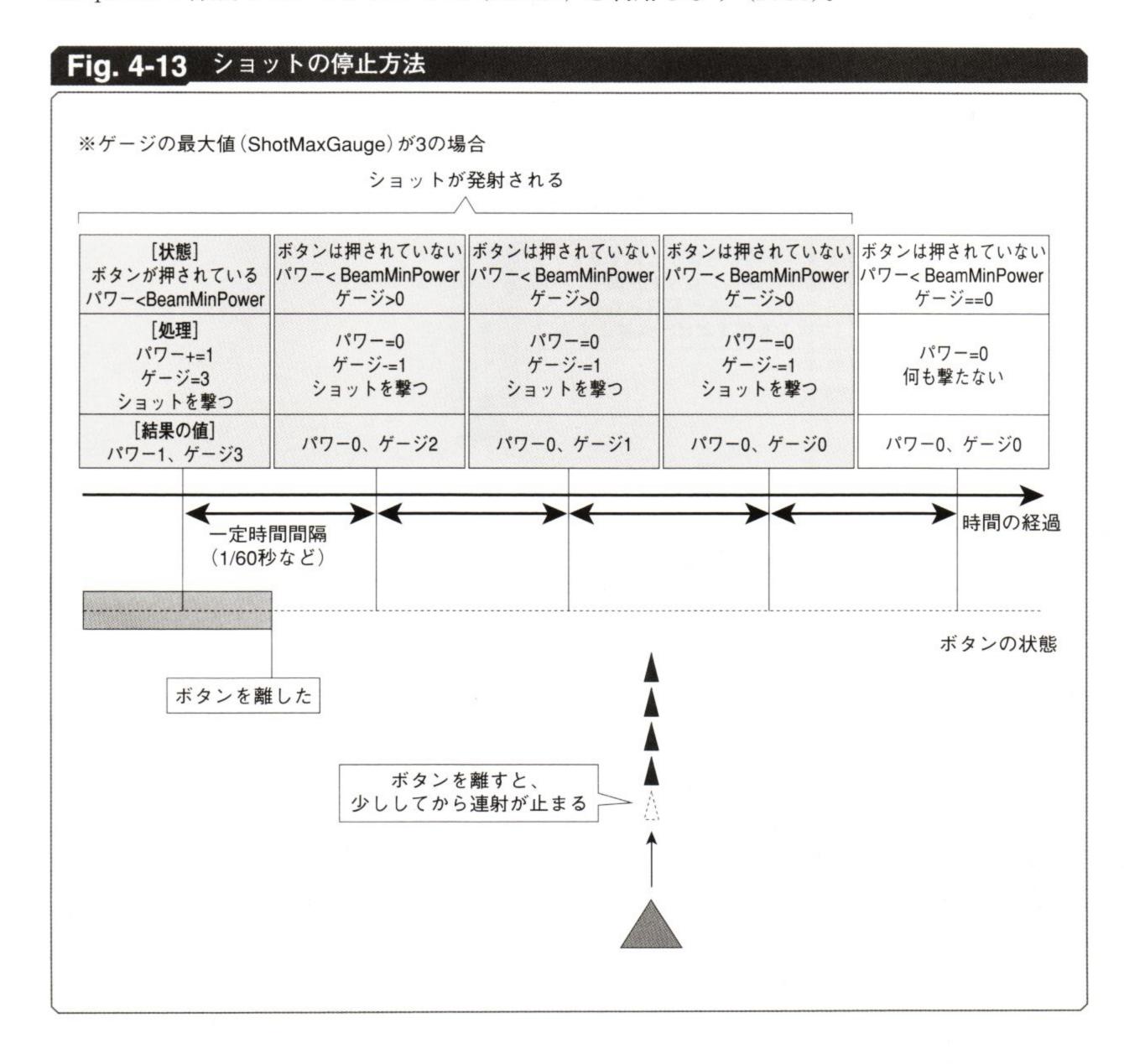



**Fig. 4-14　ビームの発射方法**

ショットが発射される　　ビームが発射される

| [状態]<br>ボタンが押されている<br>パワー<BeamMinPower | ボタンが押されている<br>パワー<BeamMinPower | ボタンが押されている<br>パワー>=BeamMinPower | ボタンが押されている<br>パワー>=BeamMinPower | ボタンは押されていない<br>ゲージ==0 |
|---|---|---|---|---|
| [処理]<br>パワー+=1<br>ゲージ=ShotMaxGauge<br>ショットを撃つ | パワー+=1<br>ゲージ=ShotMaxGauge<br>ショットを撃つ | ゲージ=0<br>ビームを撃つ | ゲージ=0<br>ビームを撃つ | パワー=0<br>何も撃たない |
| [結果の値]<br>パワーBeamMinPower-2<br>ゲージShotMaxGauge | パワーBeamMinPower-1<br>ゲージShotMaxGauge | パワーBeamMinPower<br>ゲージ0 | パワーBeamMinPower<br>ゲージ0 | パワー0<br>ゲージ0 |

時間の経過

一定時間間隔
（1/60秒など）

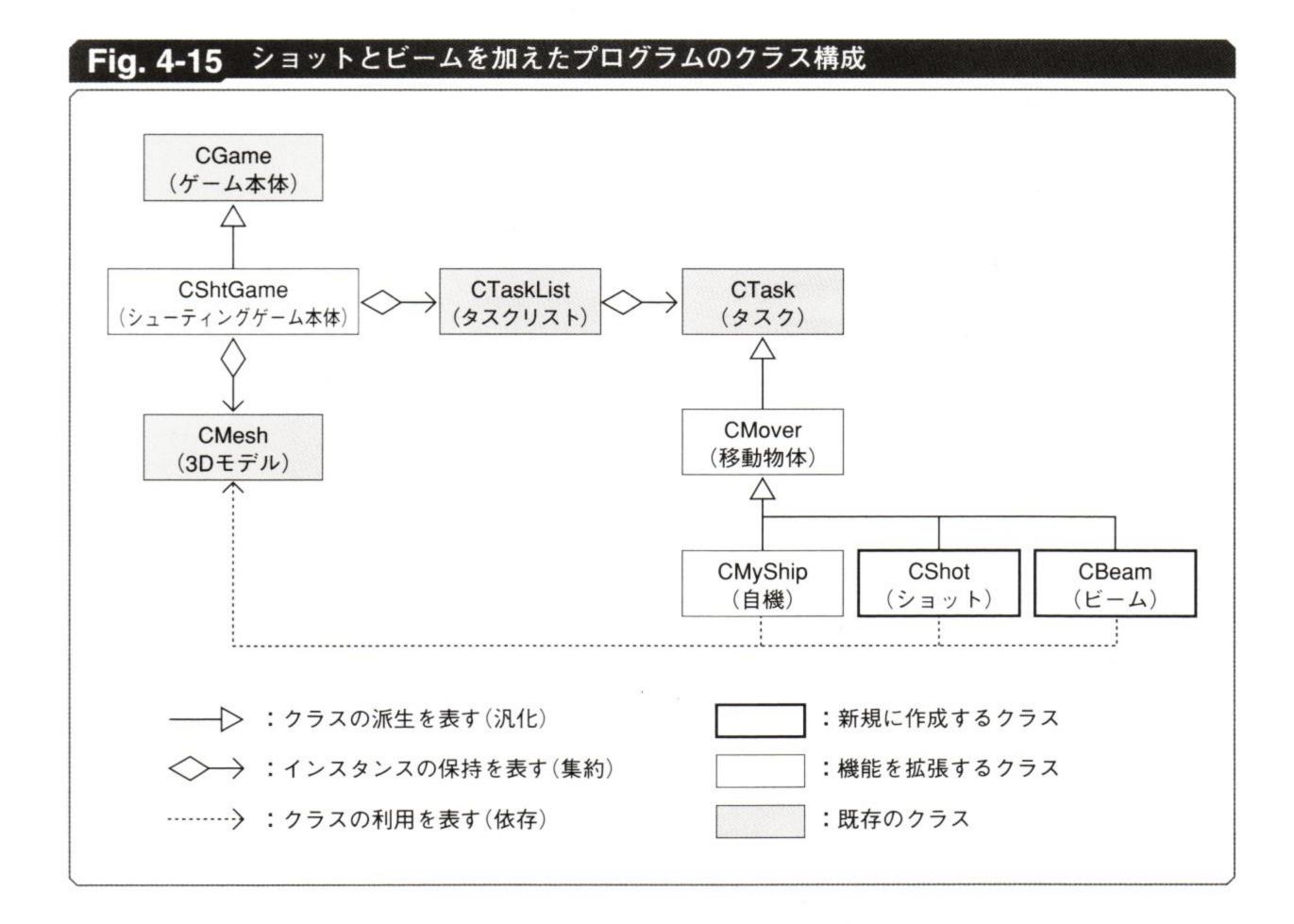
ボタンの状態

ボタンを離した

ボタンを押しっぱなしにすると、
最初に少しショットが出て、
次にビームが発射される
（ボタンを離すとビームは止まる）

**Fig. 4-15 ショットとビームを加えたプログラムのクラス構成**

CGame（ゲーム本体）
CShtGame（シューティングゲーム本体）
CTaskList（タスクリスト）
CTask（タスク）
CMesh（3Dモデル）
CMover（移動物体）
CMyShip（自機）
CShot（ショット）
CBeam（ビーム）

：クラスの派生を表す（汎化）
：インスタンスの保持を表す（集約）
：クラスの利用を表す（依存）
：新規に作成するクラス
：機能を拡張するクラス
：既存のクラス

## 自機にショットやビームを撃たせる

ショットやビームを撃つには、自機の移動処理にショットやビームの発射処理を追加します。

1フレームごとにショットやビームを発射すると、ショットやビームの密度が高くなりすぎることがあります。その場合は、4フレームに1発や2フレームに1発といった具合に、数フレームおきに発射することによって、密度を調整することができます。

List 4-11は、自機クラス（CMyShip）にショットおよびビームを撃つ処理を追加したものです。

**List 4-11 自機のクラス（MyShip.h、MyShip.cpp）**

```
// 自機のクラス
// ショットとビームに関する変数や関数を追加
class CMyShip : public CMover {
protected:
```

```
    // ショットのゲージと待機時間
    int ShotGauge, ShotTime;

    // ビームのパワーと待機時間
    int BeamPower, BeamTime;

    // ショット、ビームの発射
    void Shot();
    void Beam();

    // ビームの消去
    void DeleteBeam();

    // ...（中略）...

};

// コンストラクタ
// ショットとビームに関する変数を初期化
CMyShip::CMyShip(float x, float y)
:   CMover(Game->MyShipList, x, y, MYSHIP_Z),
    Roll(0),
    ShotGauge(0), ShotTime(0), BeamPower(0), BeamTime(0)
{}

// 移動
bool CMyShip::Move()
{
    // ...（中略）...

    // ビームのパワーとショットのゲージ
    static const int BeamMinPower=15, ShotMaxGauge=15;

    // ショットとビームの待機時間
    static const int MaxShotTime=4, MaxBeamTime=2;

    // ショットとビームの発射
    if (is->Button[0]) {

        // パワーが足りないとき：ショットを発射
        if (BeamPower<BeamMinPower) {
            BeamPower++;
            ShotGauge=ShotMaxGauge;
            if (ShotTime==0) Shot();
        }
```

```
        // パワーが足りているとき：ビームを発射
        else {
            ShotGauge=0;
            if (BeamTime==0) Beam();

            // ビーム発射中はスピードを遅くする
            vx*=0.5f;
            vy*=0.5f;
        }
    } else {

        // ボタンを離したとき：ビームを消す
        DeleteBeam();
        BeamPower=0;

        // ゲージが残っているとき：ショットを発射
        if (ShotGauge>0) {
            if (ShotTime==0) Shot();
            ShotGauge--;
        }
    }

    // ショットとビームの待機時間を更新
    // 待機時間は数フレームおきに発射するために使用
    ShotTime=(ShotTime+1)%MaxShotTime;
    BeamTime=(BeamTime+1)%MaxBeamTime;

    // ...（中略）...

}
```

## ショットを飛ばす

ショットを飛ばすには、まず発射方向と速さを決めます。そしてsin関数やcos関数を用いて、速度のX成分とY成分を計算します。

自機と同様に、ショットに関しても移動や描画の処理が必要です。移動はフレームごとに行います。計算した速度を座標に加算することによって、座標を更新します。また、ショットの発射音も再生すると迫力が増すでしょう。

ショットが画面外に飛び出してしまったら、ショットを消去しなければなりません。ショットの座標と画面端の座標を比べて、ショットが画面外に出たかどうかを判定します

(Fig. 4-16)。

ショットの中心からX軸方向またはY軸方向の端までの長さを、ここではショットのサイズとします。サイズをショットよりも少し大きめにとっておき、Fig. 4-16のような条件式で座標を判定すれば、ショットが完全に画面から出たことを検出することができます。この判定処理は、弾や敵といった他の移動物体にも適用できます。

複数のショットを同時に発射すると、迫力が増します。あるいは、パワーアップしたときにショットが複数になる、というゲームデザインもあるでしょう。本書のサンプルでは、3発のショットを同時に3方向へ発射しています(Fig. 4-17)。

以上の処理を終えたら、ショットを描画します。描画処理は基本的にフレームごとに行います。

List 4-12は、ショットのプログラムです。

**Fig. 4-16 移動物体が画面外に出たかどうかを判定する**

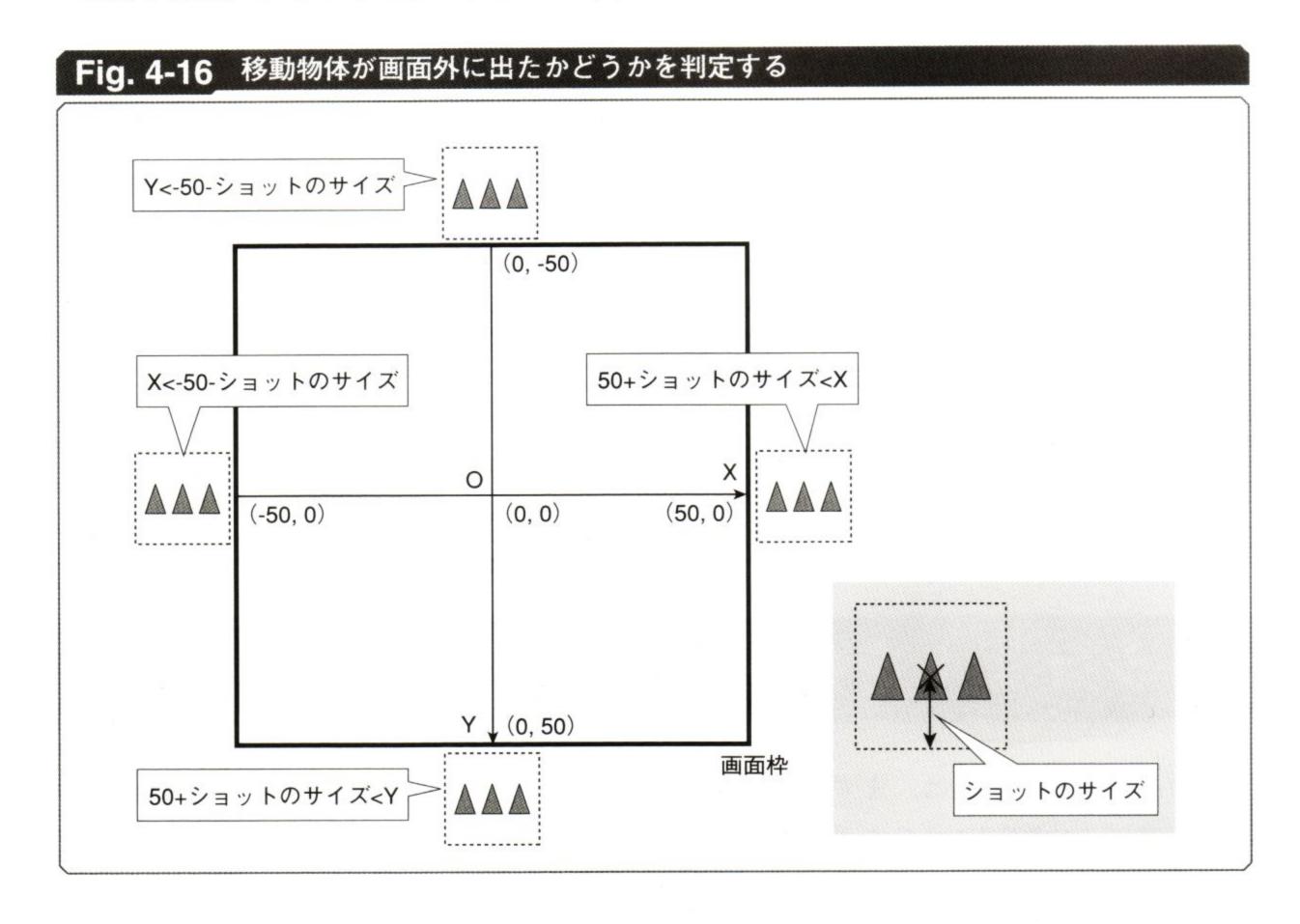
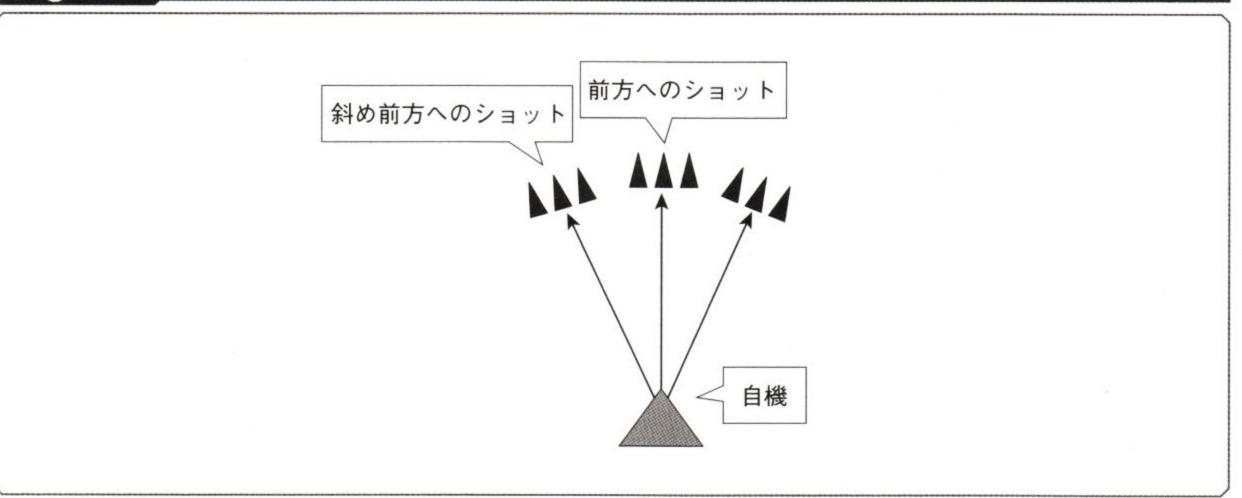



**Fig. 4-17 ショットの発射**

**List 4-12 ショットのクラス（MyShip.h、MyShip.cpp）**

```
// ショットのクラス
// 移動物体クラス（CMover）から派生
class CShot : public CMover {
protected:

    // 速度、回転角度
    float VX, VY, Yaw;

public:

    // new演算子とdelete演算子
    // ショットのタスクリスト（ShotList）を指定
    void* operator new(size_t t) {
        return operator_new(t, Game->ShotList);
    }
    void operator delete(void* p) {
        operator_delete(p, Game->ShotList);
    }

    // コンストラクタ、移動、描画
    CShot(float x, float y, float dir);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CShot::CShot(float x, float y, float dir)
:   CMover(Game->ShotList, x, y, WEAPON_Z),
```

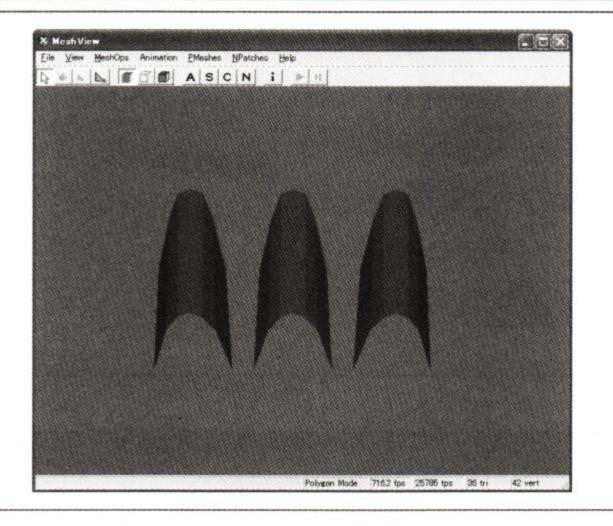
```
    Yaw(dir+0.25f)
{
    // ショットの速さは一定値（2.4f）
    // 方向は引数dirで指定
    // 例えば0は0度、0.5は180度、1.0は360度を表す
    // dirをcos関数やsin関数で使用する際にはラジアンに変換
    VX=2.4f*cos(D3DX_PI*2*dir);
    VY=2.4f*sin(D3DX_PI*2*dir);
}

// 移動
bool CShot::Move() {
    X+=VX;
    Y+=VY;

    // ショットが画面外に出たかどうかを判定し、
    // 出た場合にはfalseを返す
    return !Out(5);
}

// 移動物体が画面外に出たかどうかの判定処理
// 引数sizeには物体のサイズを与える
bool CMover::Out(float size) {
    return
        X<-50-size || 50+size<X ||
        Y<-50-size || 50+size<Y;
}

// 描画
void CShot::Draw() {
    Game->MeshShot->Draw(
        X, Z, -Y, 1, 1, 1, 0, Yaw, 0, TO_ZYX, 1, false);
}

// ショットの発射
void CMyShip::Shot() {

    // 3方向にショットのタスクを生成
    // 第1および第2引数はショットの出現位置を表し、
    // 第3引数は発射角度を1.0=360度として表す
    new CShot(X, Y, 0.70f);
    new CShot(X, Y, 0.75f);
    new CShot(X, Y, 0.80f);

    // ショットの発射音を再生
    // Chapter 2のCSoundクラスを使用（P. 42）
```

```
        Game->SEShot->Play();
    }
```

## ショットを派手に見せる

ショットの3Dモデルは、実は3個のショット片を並べて構成されています（Fig. 4-18）。画面上はショットが3つ発射されているように見えても、使用されているタスクは1つです。

多くのショット片を描くと画面がにぎやかになります。1個のショット片で1個のショットを構成することもできるのですが、この方法ならばショットのタスク数が増えないので、見た目を派手にしつつも移動処理や描画処理を呼び出すコストを軽くすることができます。この手法は多くのシューティングゲームで使われています。

Fig. 4-18 ショットの3Dモデル

# ビームを放つ

ショットとは違って、ビームは常に自機の正面へ直線状に発射されます。これは、折れ曲がらずに直線になっていた方が、ビームらしく見えるからです（Fig. 4-19）。

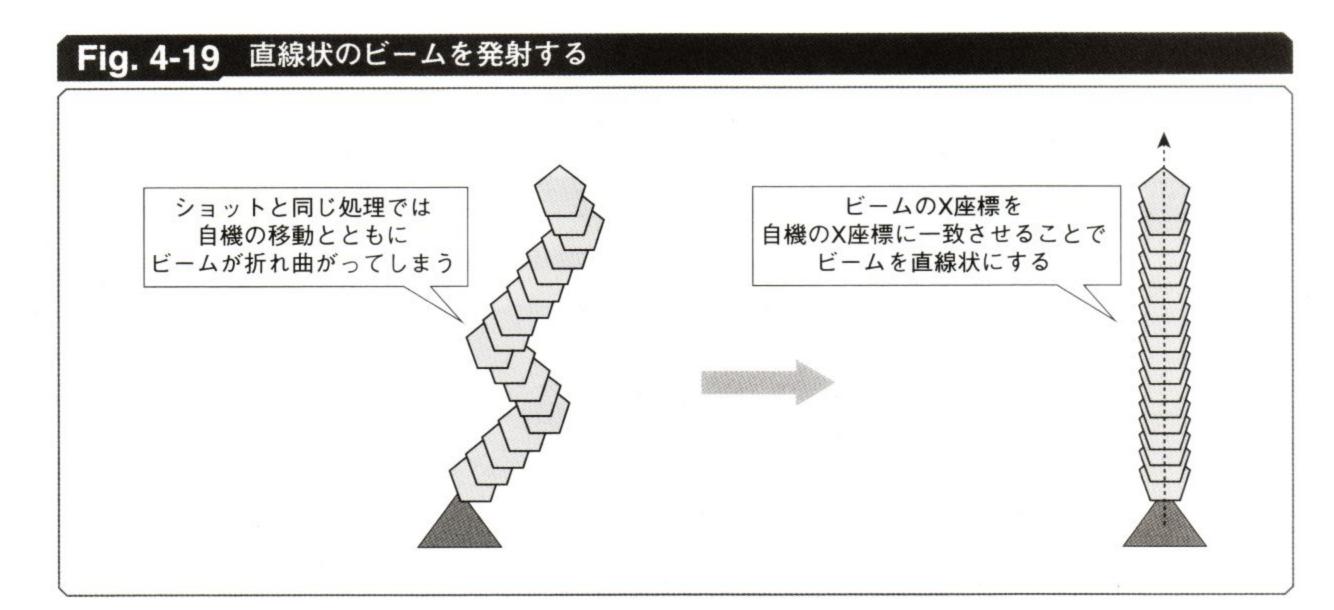
ビームを直線状にするには、常にビームのX座標を自機のX座標に合わせます。すると、ビームは自機の真正面へ一直線上に配置されます。

ビームは画面上方へ進んでいきます。これはフレームごとにビームのY座標に速度を加算することによって表現できます。ショットと同様に、画面外に出たときにはビームを消去します。移動を終えたら、ビームを描画します。

ビームを発射するときには、効果音も再生するとよいでしょう。また、ボタンを離したときにはビームを止めます。

本書のサンプルでは、ビームを多数のビーム片から構成しています。各ビーム片は独立したタスクです。効果音はビーム片を生成する際に再生しています。ビーム発射中は次々とビーム片を生成するので、効果音が繰り返し再生されます。ビームを止めるときには、ビームのタスクをすべて消去します。

List 4-13は、ビームのプログラムです。基本的な構成はショットのプログラムと同様です。

**Fig. 4-19 直線状のビームを発射する**

**List 4-13 ビームのクラス（MyShip.h、MyShip.cpp）**

```
// ビームのクラス
// CMoverクラスから派生
class CBeam : public CMover {
public:

    // 自機を指す変数
    // ビームのX座標を自機のX座標に合わせるために使用
    CMyShip* MyShip;

    // new演算子とdelete演算子
    // タスクリストにはBeamListを指定
    void* operator new(size_t t) {
        return operator_new(t, Game->BeamList);
    }
    void operator delete(void* p) {
```

```
        operator_delete(p, Game->BeamList);
    }

    // コンストラクタ
    // 引数のmyshipは、ビームを発射した自機を表す
    CBeam(CMyShip* myship, float y);

    // 移動、描画
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CBeam::CBeam(CMyShip* myship, float y)
:   CMover(Game->BeamList, myship->X, y, WEAPON_Z),
    MyShip(myship)
{}

// 移動
bool CBeam::Move() {

    // ビームのX座標を自機のX座標に合わせる
    if (MyShip) {
        X=MyShip->X;
    }

    // 自機が消去されたときはビームも消去する
    else {
        return false;
    }

    // ビームの座標を更新して画面上方に移動させる
    Y-=6.0f;

    // ビームが画面外に出た場合にはfalseを返す
    return !Out(8);
}

// 描画
void CBeam::Draw() {
    Game->MeshBeam->Draw(
        X, Z, -Y, 1, 1, 1, 0, 0, 0, TO_ZYX, 1, false);
}

// ビームの発射
void CMyShip::Beam() {
```

```
        // ビームのタスクを生成
        new CBeam(this, Y);

        // ビームの発射音を再生
        Game->SEBeam->Play();
    }

    // ビームの消去
    // ボタンを離したときにビームを止めるために使用
    void CMyShip::DeleteBeam() {
        for (CTaskIter i(Game->BeamList); i.HasNext(); ) {
            if (((CBeam*)i.Next())->MyShip==this) i.Remove();
        }
    }
```

## ショットとビームに関する初期化とデータのロード

ショットとビームの最後の仕上げです。ショットとビームを追加したので、ゲーム本体に処理を追加する必要が出ました。発射音に使うサウンドのロード、3Dモデルのロード、タスクリストの初期化を行います。

List 4-14は、ゲーム本体のプログラムに、ショットとビームに関する初期化とロードの処理を加えたものです。

**List 4-14　ショットとビームに関する初期化とデータのロード（Main.h、Main.cpp）**

```
// ゲーム本体のクラス
class CShtGame : public CGame {
protected:

    // ...（中略）...

    // サウンドのロード
    CSound* NewSound(string file);

public:

    // ...（中略）...

    // 3Dモデル（自機・ショット・ビーム）
    CMesh *MeshSauce, *MeshShot, *MeshBeam;
```

```
    // サウンド（ショット・ビーム）
    CSound *SEShot, *SEBeam;

    // タスクリスト（自機・ショット・ビーム）
    CTaskList *MyShipList, *ShotList, *BeamList;
};

// コンストラクタ
CShtGame::CShtGame()
:   CGame("紫雨 (MURASAME)", true, false, true)
{
    // ...（中略）...

    // 3Dモデルの初期化（自機・ショット・ビーム）
    LPDIRECT3DDEVICE9 device=Graphics->GetDevice();
    MeshSauce=NewMesh("sauce");
    MeshShot=NewMesh("shot");
    MeshBeam=NewMesh("beam");

    // サウンドの初期化（ショット・ビーム）
    SEShot=NewSound("shot");
    SEBeam=NewSound("beam");

    // タスクリストの初期化（自機・ショット・ビーム）
    MyShipList=new CTaskList(sizeof(CMyShip), 10);
    ShotList=new CTaskList(sizeof(CShot), 100);
    BeamList=new CTaskList(sizeof(CBeam), 100);
}

// サウンドのロード
// Chapter 2で作成したCSoundクラスを使用（P. 42）
CSound* CShtGame::NewSound(string file) {
    string path=ExePath+"..¥¥Sound¥¥";
    CSound* p=new CSound();
    p->LoadFromFile(path+file+".wav");
    return p;
}
```

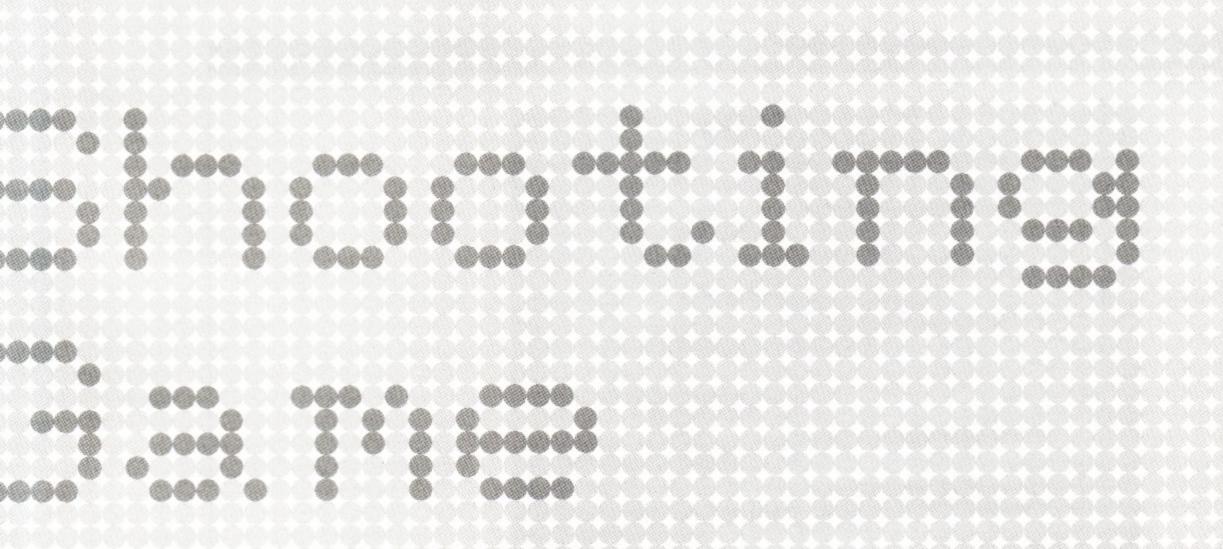
## >>Chapter 4のまとめ

本章では、自機の動きに加えて、ショットとビームの発射について解説しました。まだ自機の処理だけで、弾も敵も出てきませんが、これから本格的なシューティングゲームに育てていきましょう。

ショットやビームに関する定数を変更すると、速度や角度を変えることができます。ショットの角度を変えたり、ビームの速度を変えたりと、いろいろ試してみてください。

# Chapter 05

# 弾

本章では敵の発射する弾に関するプログラムを作ります。弾の移動と表示、自機と弾との当たり判定処理、自機の破壊、そして弾幕の作り方について説明します。
ここまで作ってしまえば、かなりシューティングゲームらしくなります。ゲームデザインやバランス調整しだいで、自機と弾だけでも楽しく遊べるゲームに仕上げることも可能です。

# 弾と当たり判定処理

自機を動かすことができるようになったら、次は弾を動かしてみるのがお勧めです。同時に、自機と弾との当たり判定処理や、弾に当たったときに自機を破壊する処理も作ります。

自機と弾、そして当たり判定処理ができれば、シューティングゲームとして最小限必要な要素がそろいます。自機を動かして弾を避けるだけでも、ゲームとして成立させることは可能です。

Fig. 5-1～3は、弾と当たり判定処理を追加したサンプルの実行画面です。プロジェクトは付録CD-ROMの「ShtGame_Bullet1」フォルダに収録しています。実行ファイルは「ShtGame_Bullet1¥Release¥ShtGame.exe」です。

サンプルでは、自機が出現するときと、自機が破壊されたときに、無敵期間（弾に接触しても破壊されない期間）を設けました。最初に、画面下部から自機が回転しながら出現します。回転中の自機は無敵です。画面上部から方向弾（一定方向に進む弾）が発射されるので、自機を操作して弾を避けます（Fig. 5-1）。弾に接触すると、自機は破壊されます（Fig. 5-2）。少し時間がたつと、自機は復活します。自機が点滅している間は無敵期間です（Fig. 5-3）。

自機を操作して弾を避けるだけですが、意外にゲームとして遊べるものになっています。今は弾が1種類だけですが、本章の後半では弾の種類を増やして、より遊べるゲームにします。

**Fig. 5-1** 自機の出現

**Fig. 5-2** 自機の破壊

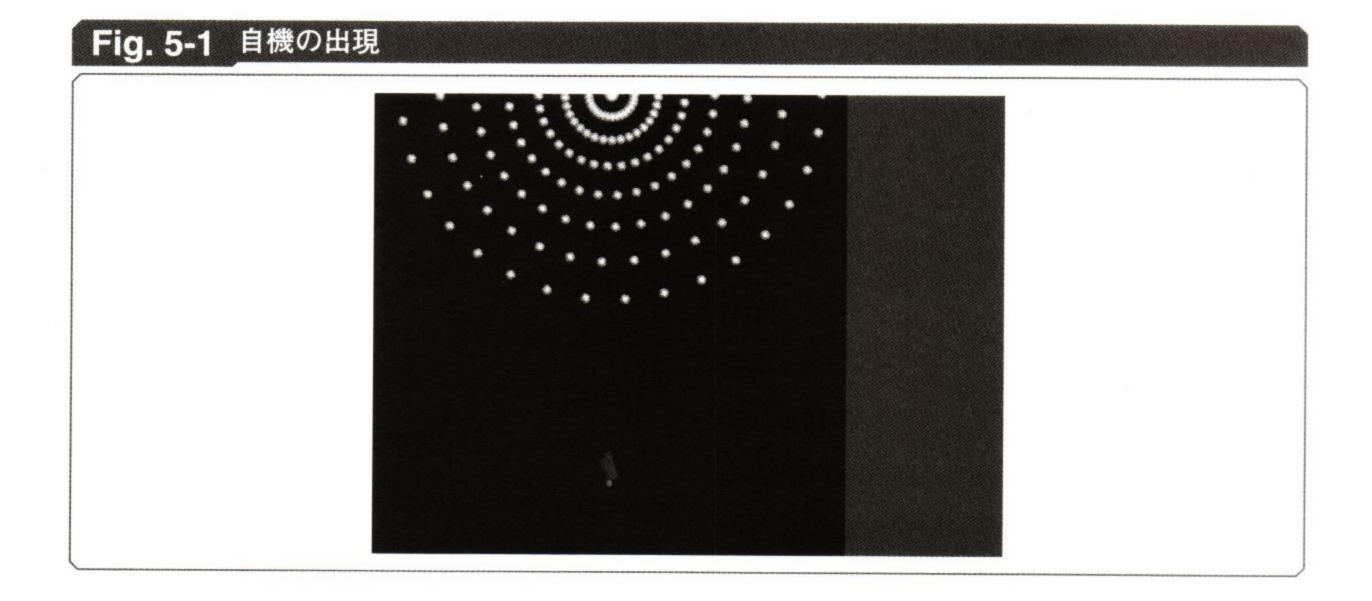
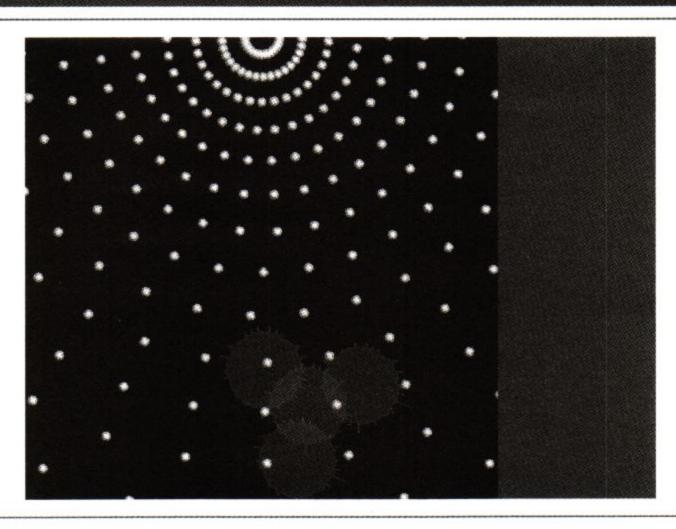
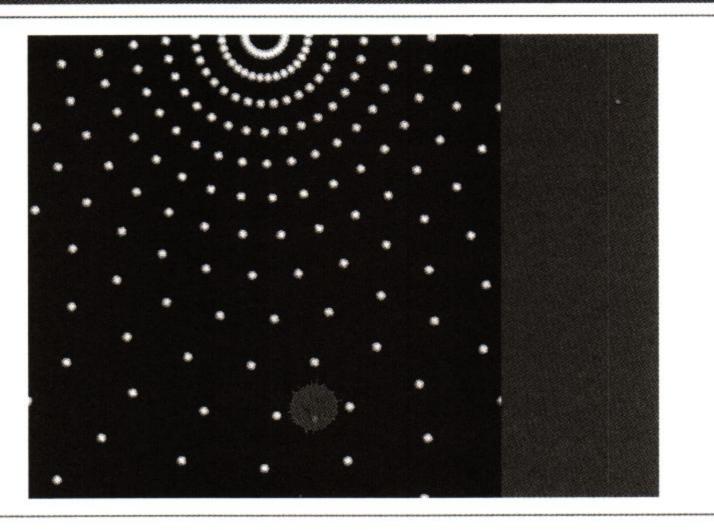
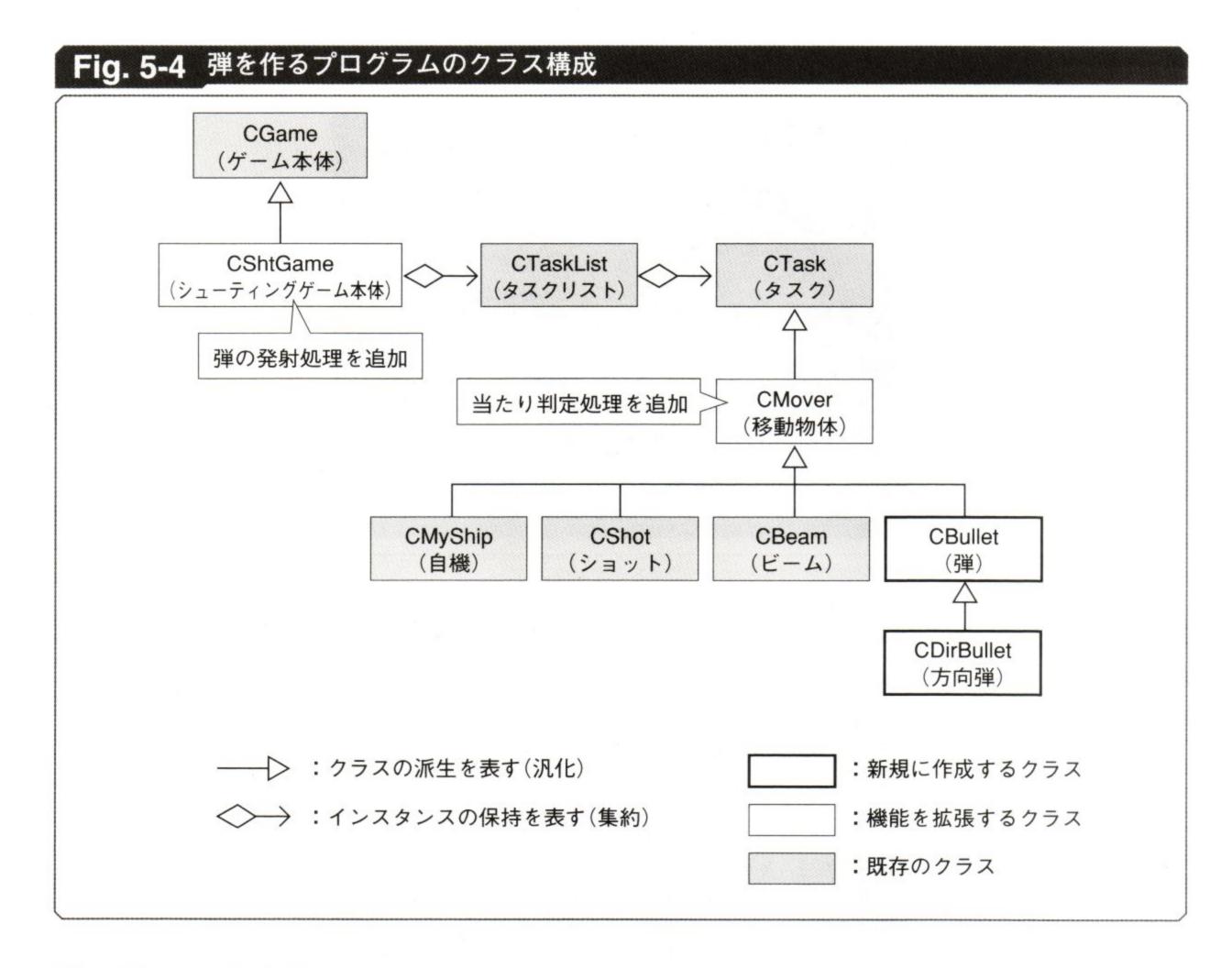



**Fig. 5-3** 自機の復活

## 弾を動かす

弾のプログラムでは、弾を作る処理と、自機と弾の当たり判定処理がポイントとなります。最初に弾を作る処理に関して説明します。

Fig. 5-4はサンプルのクラス構成です。弾の共通機能をまとめたクラス（CBullet）と、方向弾を表すクラス（CDirBullet）を新規に作成します。また、ゲーム本体クラス（CShtGame）や移動物体クラス（CMover）を拡張します。

各クラスの役割は次のとおりです。

**Fig. 5-4** 弾を作るプログラムのクラス構成

### CBulletクラス

弾の共通機能をまとめたクラスです。さまざまな弾の基底クラスになります。CMoverクラスから派生します。

### CDirBulletクラス

方向弾のクラスです。CBulletクラスから派生します。

### CMoverクラス

移動物体のクラスです。当たり判定処理のための関数を追加します。

### CShtGameクラス

シューティングゲーム本体のクラスです。弾の3Dモデルやタスクリストを初期化する処理と、弾を発射する処理を追加します。

## 弾を作る

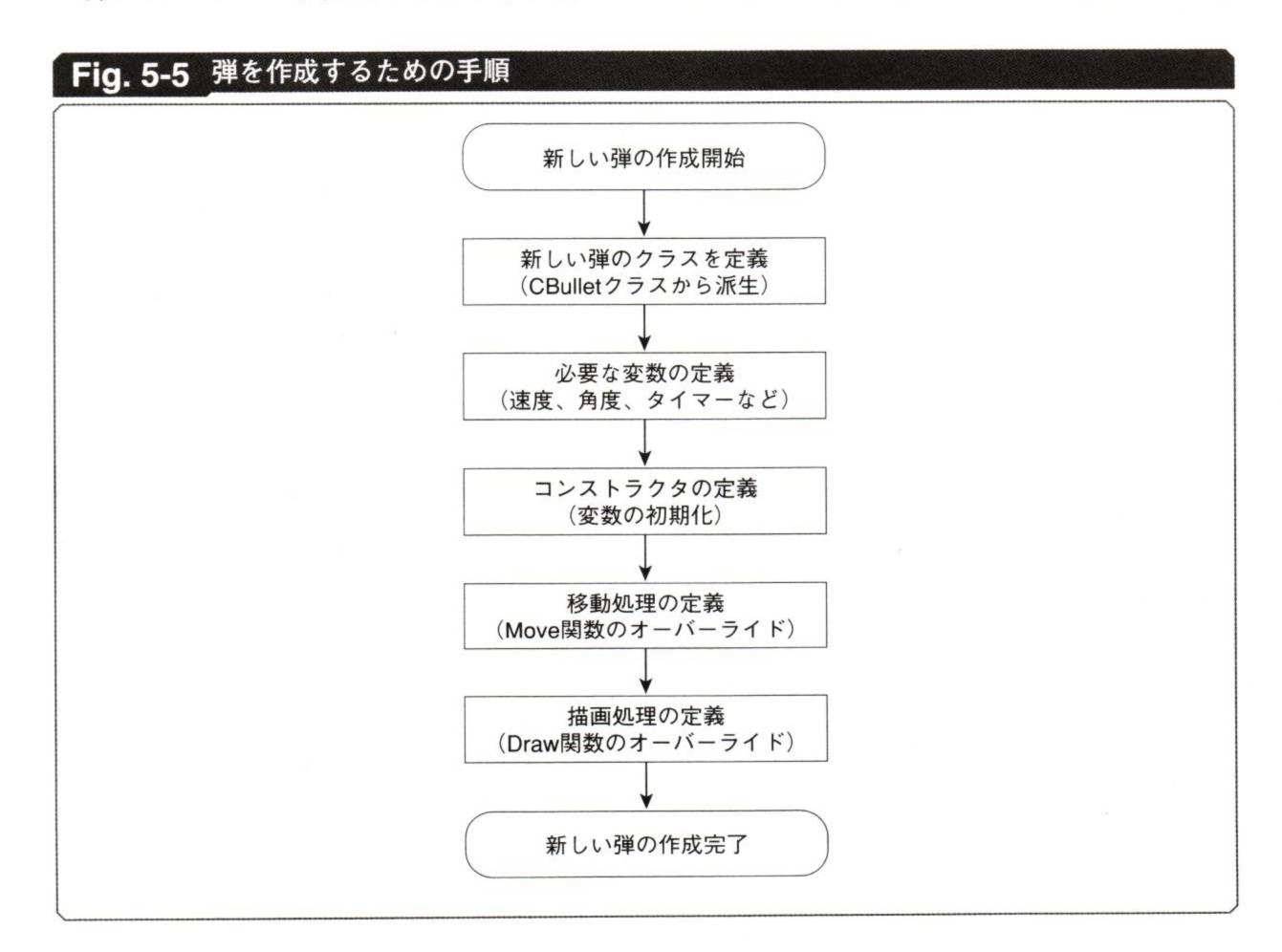
弾を作るには、弾を移動する処理と、弾を描画する処理を作成する必要があります。ゲームプログラムの構造に応じて実装の詳細は微妙に変わりますが、移動と描画という基本的な構成は共通です。

ここでは前章で作成した移動物体クラス(CMover)をベースにして、弾のクラスを作ります。移動物体クラスは自機・弾・敵といった移動物体に共通の機能をまとめたクラスでした(P. 109)。

移動物体クラスは、座標を保持する変数、移動を行う関数、および描画を行う関数を備えています。まず、この移動物体クラスに当たり判定処理を追加しておきましょう。

弾を作るには、これに加えて回転角度や、角度が変化する速度といった変数が必要です。そこで、移動物体クラスの派生クラスとして、弾クラス(CBullet)を定義し、これらの必要な変数をメンバ変数として定義します。また、移動物体クラスの移動関数と描画関数をオーバーライドして、さまざまな弾に共通する移動や描画の処理を記述します。

弾にはいろいろな種類があります。弾クラスにはこれらに共通する処理を記述するにと

**Fig. 5-5** 弾を作成するための手順

どめ、個々の弾を表すクラスは弾クラスから派生させます。Fig. 5-5は、こうした個々の弾を作成するまでの手順をまとめたものです。

## 移動物体に当たり判定を与える

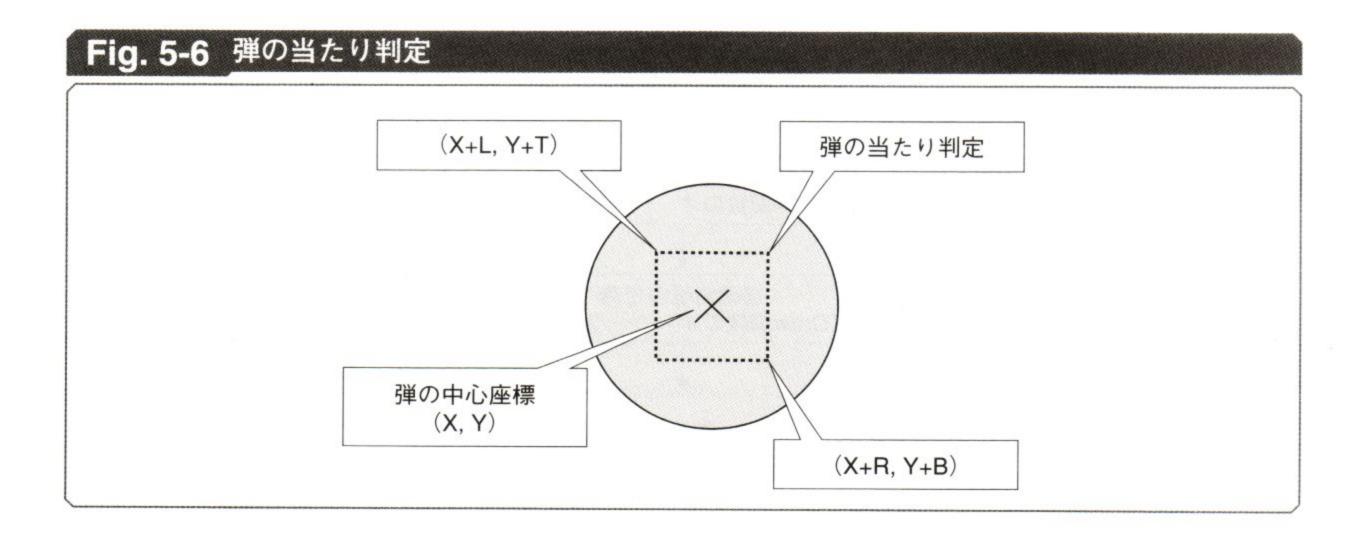
弾は自機に当たったかどうかを判定する必要があります。当たり判定処理は多くの移動物体に共通するものなので、移動物体クラス（CMover）を拡張して、当たり判定処理の機能を追加することにしました。

まず、当たり判定の領域を指定するための変数を追加します。当たり判定の左端・右端・上端・下端の座標を保持するようにします。これらは、弾の中心座標に対する相対座標です。それぞれの値を0に近くすると当たり判定が小さくなり、絶対値を大きくすると当たり判定が大きくなります。当たり判定は左上が（X+L, Y+T）、右下が（X+R, Y+B）の矩形になります（Fig. 5-6）。

当たり判定処理を行う関数には、2つのパターンを用意しました。1つめは、他の移動物体との間で判定処理を行う関数です。2つの移動物体がそれぞれ持っている当たり判定が重なっているかどうかを判定します。

2つめも、他の移動物体との間で判定処理を行う関数ですが、引数で当たり判定を指定できる点が異なります。この関数は、かすり（P. 346）などのように特別の当たり判定を使いたいときに便利です。なお、当たり判定の条件式に関しては、Chapter 1を参照してください（P. 8）。

List 5-1は、当たり判定処理を追加した移動物体のプログラムです。移動物体クラスのコンストラクタは、当たり判定を指定するものと、指定しないものの2種類を用意しました。こうしておけば、当たり判定のない移動物体を作成するときには当たり判定の指定を省略できます。

**Fig. 5-6 弾の当たり判定**

**List 5-1　移動物体のクラス（Mover.h）**

```
// 移動物体のクラスはタスククラス（CTask）から派生する
class CMover : public CTask {
public:

    // 座標
    float X, Y, Z;

    // 当たり判定の左端・右端・上端・下端の相対座標
    float L, T, R, B;

    // コンストラクタ（当たり判定を指定する）
    CMover(
        CTaskList* task_list, float x, float y, float z,
        float l, float t, float r, float b
    )
    :   CTask(task_list), X(x), Y(y), Z(z),
        L(l), T(t), R(r), B(b)
    {}

    // コンストラクタ（当たり判定を指定しない）
    CMover(CTaskList* task_list, float x, float y, float z)
    :   CTask(task_list), X(x), Y(y), Z(z),
        L(0), T(0), R(0), B(0)
    {}

    // 当たり判定処理（他の移動物体を指定）
    bool Hit(CMover* m) {
        return
            X+L<m->X+m->R && m->X+m->L<X+R &&
            Y+T<m->Y+m->B && m->Y+m->T<Y+B;
    }

    // 当たり判定処理（当たり判定を指定）
    bool Hit(CMover* m, float l, float t, float r, float b) {
        return
            X+l<m->X+m->R && m->X+m->L<X+r &&
            Y+t<m->Y+m->B && m->Y+m->T<Y+b;
    }

    // ...（中略）...

};
```

## 弾に共通する処理をまとめる

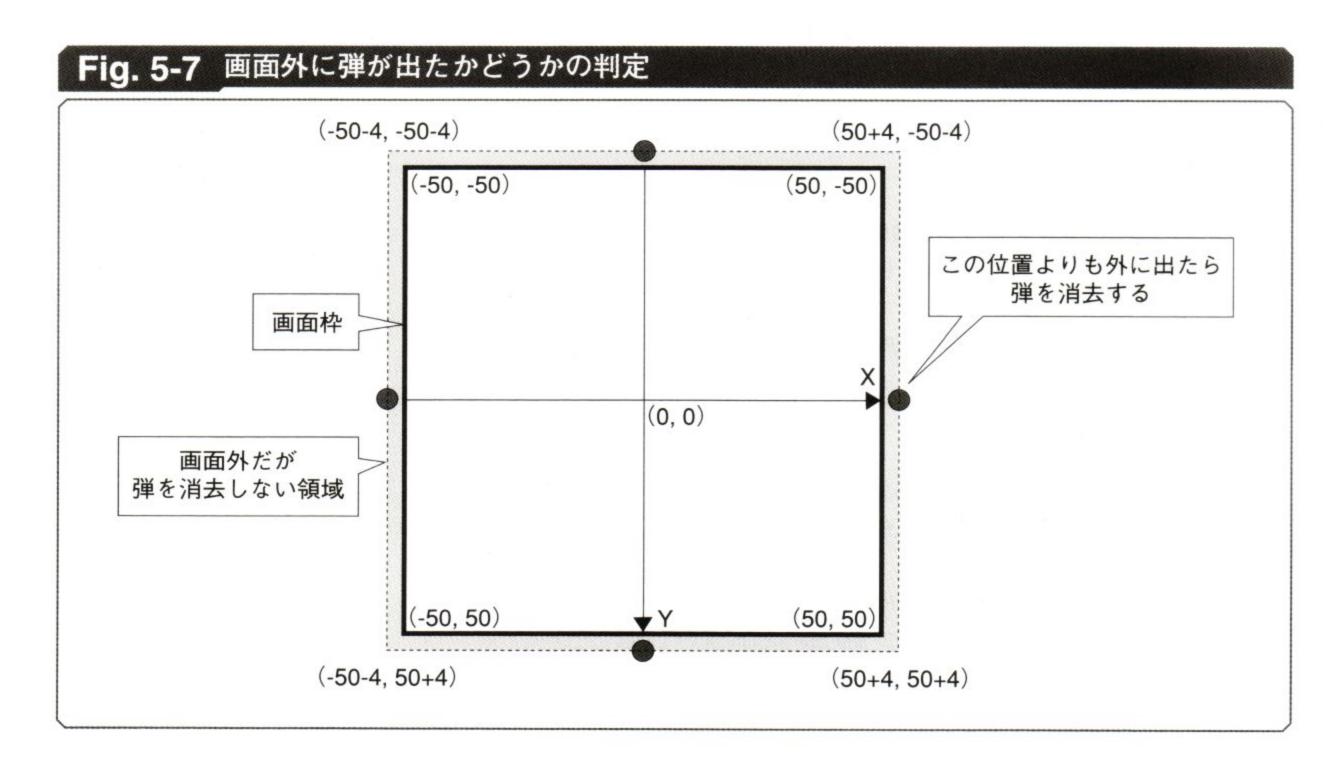
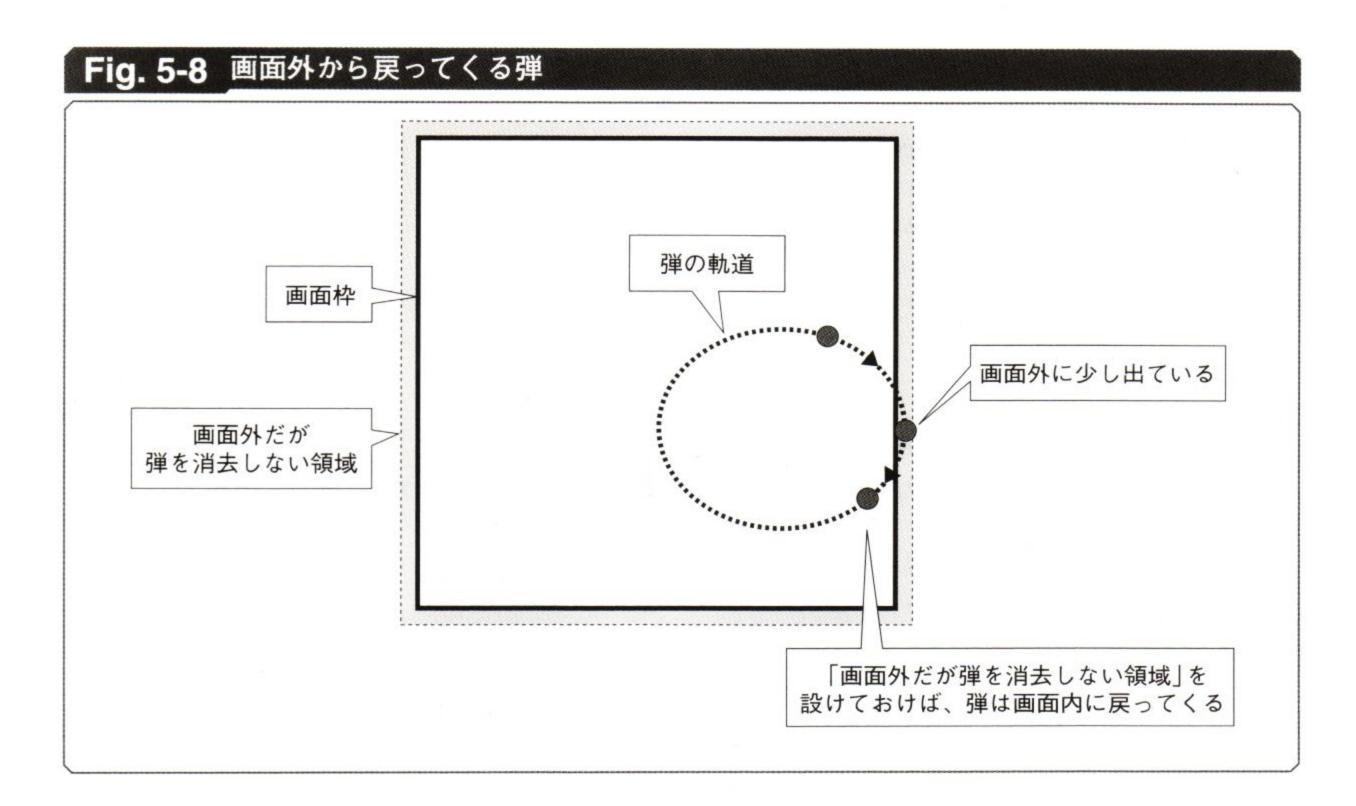
自機のショットと同様に、画面外に出た弾は消去します。Fig. 5-7のようにして、弾が画面外に出たかどうかをチェックします。弾の大きさを考慮して、弾が動ける範囲を画面枠よりも広めにとることがポイントです。こうすれば、弾が画面内で突然消えることなく、画面外に出てから自然に消えます。

また、例えば円を描いて飛ぶような弾の場合、動ける範囲を広めにとっておけば、少しなら、画面の外に出てしまっても消去されずに再び画面内に戻ってくることができます(Fig. 5-8)。

弾の共通の動きとして、弾の向きを回転させる処理も記述します。ただし、より具体的な弾の動きは、方向弾や狙い撃ち弾といった派生クラスで実装します。

List 5-2は、弾に共通の処理をまとめたクラスです。コンストラクタ内でCMoverコンストラクタを呼び出しますが、このときの引数を変更すると当たり判定の大きさを調整することができます。弾が画面外に出たかどうかを判定するときには、移動物体クラス(CMover)のOut関数を使用しています(P. 116)。

**Fig. 5-7 画面外に弾が出たかどうかの判定**

Fig. 5-8 画面外から戻ってくる弾

List 5-2 弾のクラス(Bullet.h、Bullet.cpp)

```
// 弾のクラス
class CBullet : public CMover {
protected:

    // 3Dモデル
    CMesh* Mesh;

    // 回転角度、角度の変化
    float Yaw, VYaw;

public:

    // 色
    int Color;

    // new演算子、delete演算子
    // タスクが属するタスクリストを引数に指定
    // BulletListは弾のタスクリスト
    void* operator new(size_t t) {
        return operator_new(t, Game->BulletList);
    }
    void operator delete(void* p) {
        operator_delete(p, Game->BulletList);
```

```
    }

    // コンストラクタ・移動関数・描画関数の宣言
    // CMeshは3Dモデルのクラス（P. 30）
    CBullet(CMesh** mesh, int color, float x, float y);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
// CMoverコンストラクタの最後の4つの引数は当たり判定を表す
CBullet::CBullet(CMesh** mesh, int color, float x, float y)
:   CMover(
        Game->BulletList, x, y, BULLET_Z,
        -0.3f, -0.3f, 0.3f, 0.3f),
    Mesh(mesh[color]), Color(color), Yaw(0), VYaw(0)
{
    // 丸い弾の場合には回転させる
    if (mesh==Game->MeshBullet) VYaw=0.01f;
}

// 移動
bool CBullet::Move() {

    // 弾を回転させる
    Yaw+=VYaw;

    // 弾が画面外に出たら消滅させる
    return !Out(4);
}

// 描画
void CBullet::Draw() {
    Mesh->Draw(X, Z, -Y, 1, 1, 1, 0, Yaw, 0, TO_ZYX, 1, false);
}
```

# 方向弾

方向弾は、発射時に指定された方向に一定の速度で飛行する弾です。シューティングゲームにおいて、最も基本的な弾の1つだといえます。

方向弾クラスは弾クラス（CBullet）から派生させます。弾クラスは、弾の回転角度を保

持していましたが、方向弾クラスには、さらに速度と加速度を追加します。例えば、X方向の速度をVX、Y方向の速度をVYで表します。加速度は同様に、AXとAYなどで表します。

方向弾クラスのコンストラクタには、弾の座標（x, y）、方向（dir）、速さ（spd）、加速度の大きさ（accel）といった引数を用意します。弾クラスから派生するので、方向弾クラスのコンストラクタは最初に弾クラスのコンストラクタを呼び出します。

次に、指定された方向と速さから、弾の速度（VX, VY）を計算します。加速度（AX, AY）も同様の方法で計算します。本書では、弾の方向は0.0〜1.0の数値で与えることにしました。0.0〜1.0が0度〜360度に相当します。これはラジアン（0〜2πが0度〜360度に相当）で指定するようにしてもかまいません。

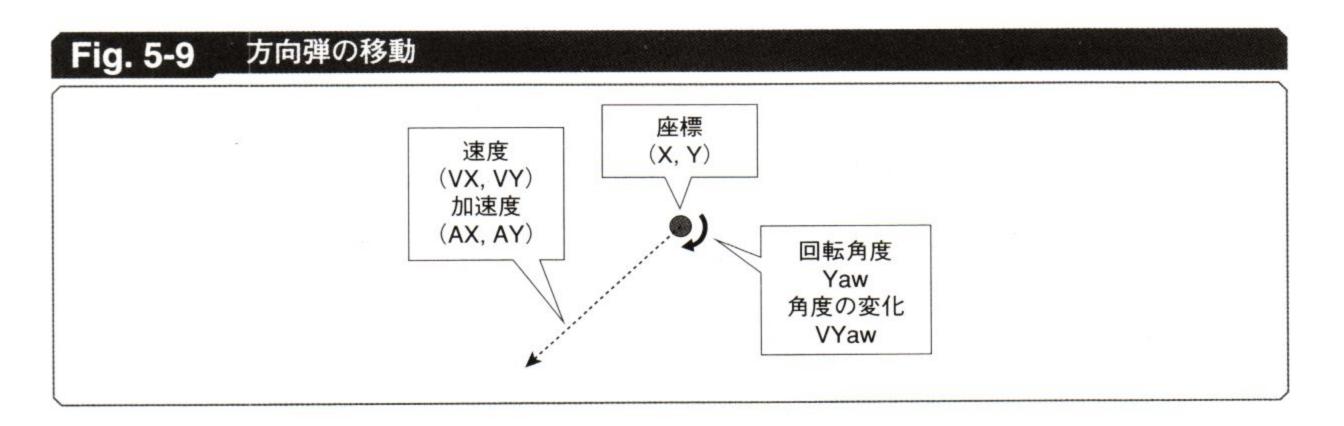
移動に関しては、弾クラスの移動処理（CBullet::Move関数）をオーバーライドします。自機クラス（CMyShip）の場合と同様に、この関数はフレームごとに（1/60秒などの一定間隔で）繰り返し実行されます。1回の呼び出しごとに弾を少しずつ動かすことによって、連続して弾が動いているように見せるのです。

座標に速度を加算し、速度に加速度を加算することによって、弾を加速させながら移動させます。コンストラクタの引数で加速度を0にすれば、一定速度で進む弾にすることもできます。

移動の最後には、弾クラスの移動処理を呼び出します。この移動処理は、弾の回転と、弾が画面外に出たかどうかの判定を行います。これはすべての弾に共通する処理です。

弾の回転は、回転角度に定数を加算することによって行っています。この様子を示したのがFig. 5-9です。弾を回転させることは必須ではありませんが、回転させた方が見た目に面白かったので、取り入れてみました。市販のシューティングゲームでも、弾を回転させているものをよく見かけます。

List 5-3は、方向弾クラスです。描画については、継承した弾クラスの描画処理（CBullet::Draw関数）をそのまま使用しています。

**Fig. 5-9 方向弾の移動**

**List 5-3　方向弾のクラス（Bullet.h、Bullet.cpp）**

```
// 方向弾のクラス
class CDirBullet : public CBullet {

    // 速度、加速度
    // VXはX方向の速度、VYはY方向の速度
    float VX, VY, AX, AY;

public:

    // コンストラクタ、移動
    CDirBullet(
        CMesh** mesh, int color, float x, float y,
        float dir, float spd, float accel);
    virtual bool Move();

};

// コンストラクタ
CDirBullet::CDirBullet(
    CMesh** mesh, int color, float x, float y,
    float dir, float spd, float accel
)
:   CBullet(mesh, color, x, y)
{
    // 速度
    float rad=D3DX_PI*2*dir, c=cos(rad), s=sin(rad);
    VX=spd*c;
    VY=spd*s;

    // 加速度
    AX=accel*c;
    AY=accel*s;

    // 回転角度（0.0～1.0が0度～360度に相当する）
    Yaw=dir+0.25f;
}

// 移動
bool CDirBullet::Move() {

    // 座標の更新
    X+=VX;
    Y+=VY;
```

```
    // 速度の更新
    VX+=AX;
    VY+=AY;

    // 弾の共通処理（回転と画面外に出たかどうかの判定）
    return CBullet::Move();
}
```

## 弾の初期化と生成

作成した弾をゲームに登場させましょう。弾に関する初期化処理と、弾を生成する処理をゲーム本体に追加します。自機の破壊に関する処理も追加しますが、これについては後述します（P. 162）。

弾のふるまいを決定するために乱数を使う場合があるので、乱数を生成する関数も用意しておきます。本書では、0以上1以下の乱数を発生するRand1関数と、-0.5以上0.5以下の乱数を発生するRand05関数を用意しました。

乱数の生成処理については乱数クラス（CRand）にまとめました。この乱数クラスは、松本眞氏、西村拓士氏らによって開発されたMersenne Twister法のCプログラムをベースにさせていただいております。大変有用なプログラムを公開していただいているオリジナル版の開発者の皆様に篤く御礼申し上げます。この乱数クラスの詳細は、付録CD-ROMの「LibUtil¥Rand.h」と「LibUtil¥Rand.cpp」をご覧ください。

List 5-4は、ゲーム本体のクラスです。移動処理では、弾の生成を8フレームごとに行います。1フレームごとに行うと、弾が多くなりすぎるからです。forループを使って、画面上部に21個の方向弾を扇状に発射します。これはいわゆる「n-way弾」です。発射の方向は、乱数を使って微妙に変化させています。

**List 5-4　ゲーム本体のクラス（Main.h、Main.cpp）**

```
// ゲーム本体のクラス
class CShtGame : public CGame {

    // ...（中略）...

    // 乱数
    CRand Rand;
    float Rand1() {
        return Rand.Real1();
```

```
    }
    float Rand05() {
        return Rand.Real1()-0.5f;
    }

    // 3Dモデル
    CMesh
        *MeshSauce, *MeshShot, *MeshBeam,
        *MeshBullet[2], *MeshNeedle[2], *MeshCrashSauce;

    // タスクリスト（弾のタスクリストを新規追加）
    CTaskList
        *MyShipList, *ShotList, *BeamList,
        *BulletList, *EffectList;
};

// コンストラクタ
CShtGame::CShtGame()
:   CGame("紫雨 (MURASAME)", true, false, false),
    MyShip(NULL), Time(0)
{
    // ...（中略）...

    // 3Dモデルの初期化（弾の3Dモデルのロード処理を追加）
    LPDIRECT3DDEVICE9 device=Graphics->GetDevice();
    MeshBeam=NewMesh("beam");
    MeshCrashSauce=NewMesh("crash_sauce");
    MeshSauce=NewMesh("sauce");
    MeshShot=NewMesh("shot");
    MeshBullet[0]=NewMesh("ikura_bullet");
    MeshBullet[1]=NewMesh("wasabi_bullet");
    MeshNeedle[0]=NewMesh("ikura_needle");
    MeshNeedle[1]=NewMesh("wasabi_needle");

    // タスクリストの初期化（弾のタスクリストを新規追加）
    BeamList=new CTaskList(sizeof(CBeam), 100);
    BulletList=new CTaskList(sizeof(CDirBullet), 2000);
    EffectList=new CTaskList(sizeof(CMyShipCrash), 2000);
    MyShipList=new CTaskList(sizeof(CRevivalMyShip), 2);
    ShotList=new CTaskList(sizeof(CShot), 100);
}

// 移動（ゲーム本体の動作）
void CShtGame::Move() {

    // 弾の生成
```

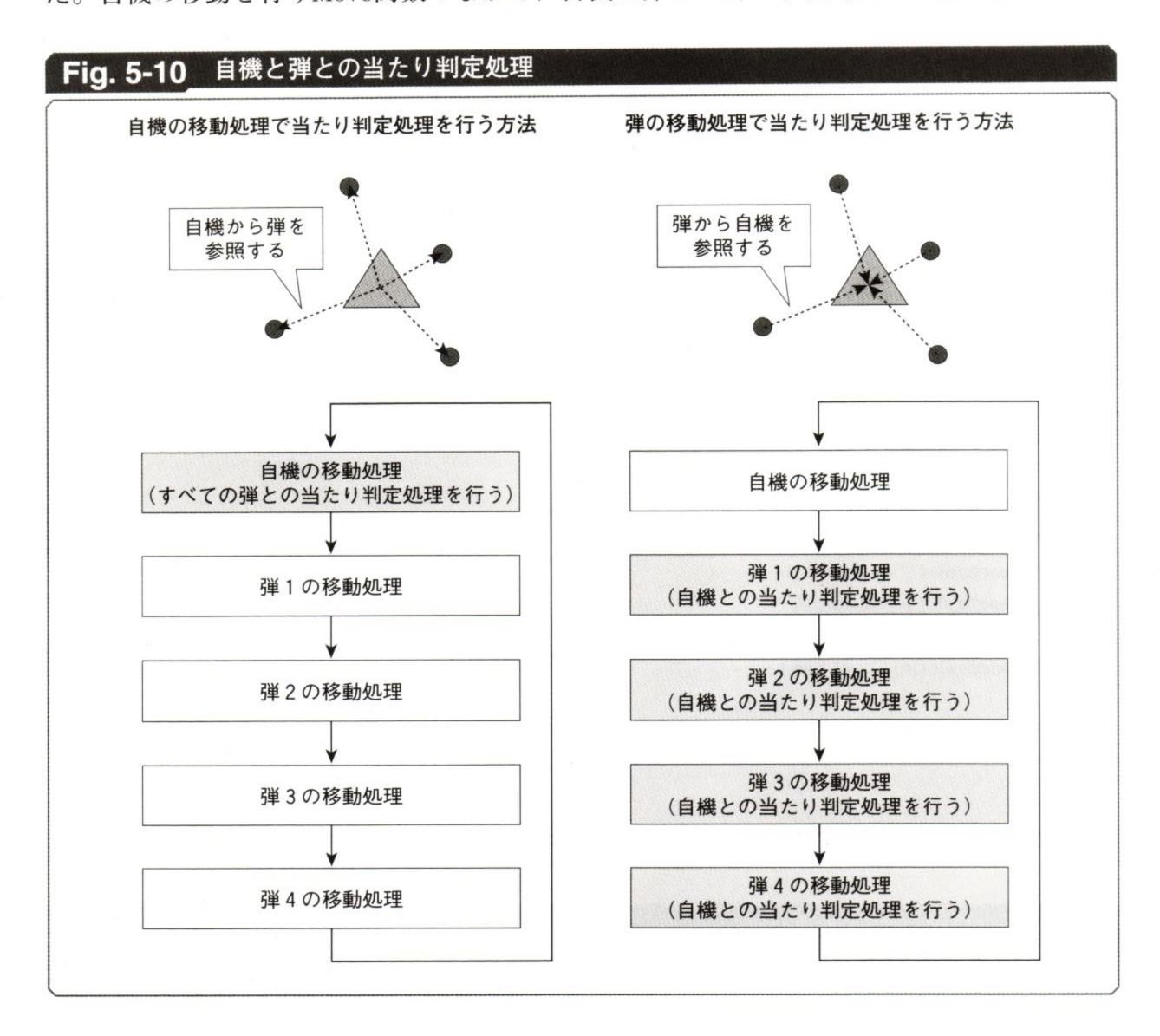
```
    if (Time%8==0) {
        float dir=Rand05()*0.06f;
        for (int i=-10; i<=10; i++) {
            new CDirBullet(
                Game->MeshBullet, 0, 0, -50,
                0.25f+dir+0.03f*i, 0.5f, 0.01f);
        }
    }
    Time++;

    // タスクの移動
    MoveTask(BulletList);
    MoveTask(BeamList);
    MoveTask(EffectList);
    MoveTask(MyShipList);
    MoveTask(ShotList);
}

// 描画
void CShtGame::Draw() {

    // ...（中略）...

    // キャラクターの描画
    DrawTask(EffectList);
    DrawTask(ShotList);
    DrawTask(BeamList);
    DrawTask(MyShipList);
    DrawTask(BulletList);

    // ...（中略）...

}
```

## 自機と弾の当たり判定処理

シューティングゲームでは、自機と弾が接触したら、自機を破壊するか、もしくは自機にダメージを与える必要があります。そのために必要なのが、自機と弾との当たり判定処理です。ここまでのプログラムで、弾を動かすことはできました。次は当たり判定処理を作成します。

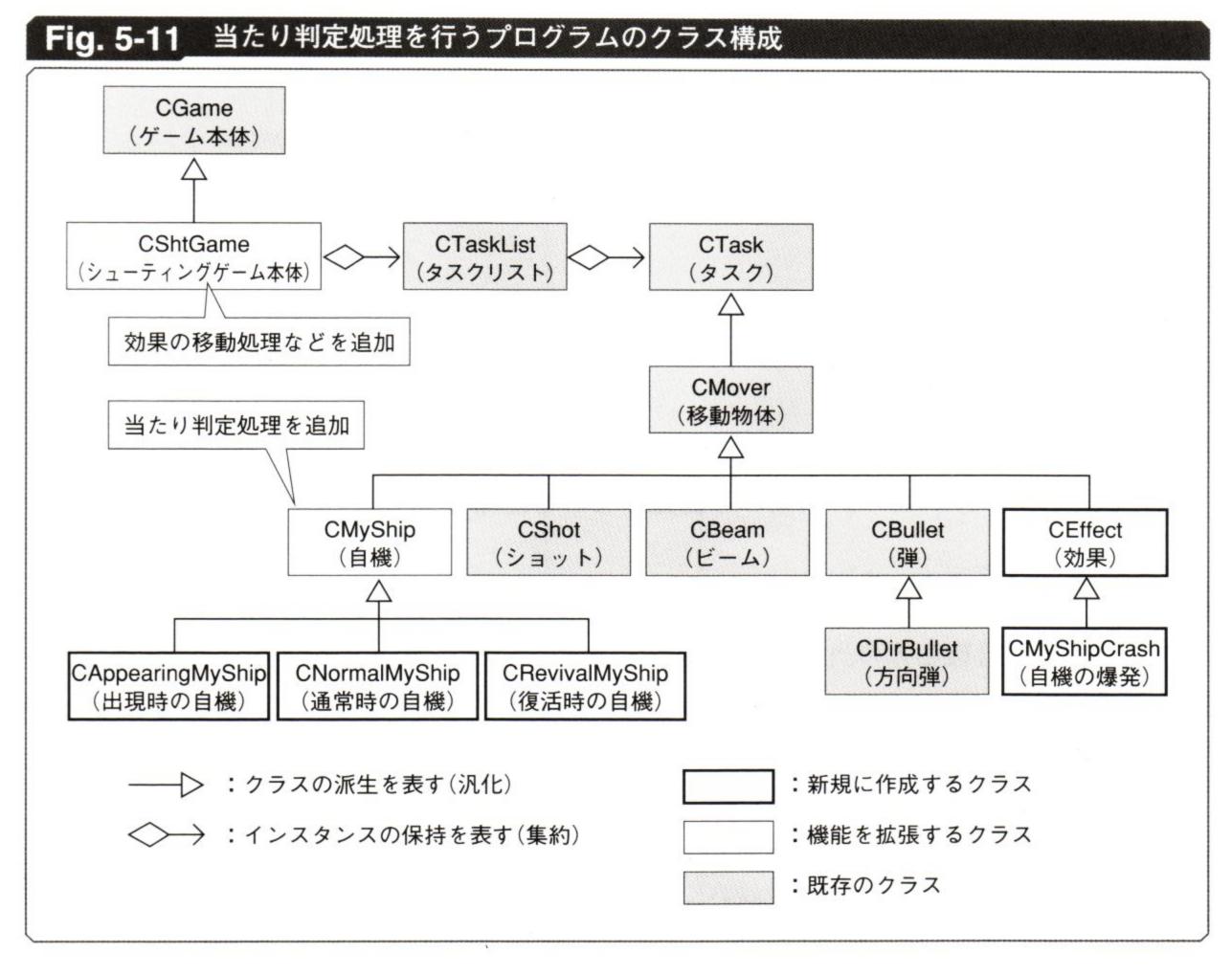
自機と弾との当たり判定処理は、自機の移動処理のなかで行う方法と、弾の移動処理のなかで行う方法とがあります。それぞれの様子をFig. 5-10に示しました。この場合、どちらの方法を使っても結果はほぼ同じなので、プログラムの構造に応じて使いやすい方法を採用するとよいでしょう。

今回はFig. 5-10の左側に示した、自機の移動処理のなかで判定を行う方法を採用しました。自機の移動を行うMove関数のなかで、自機と弾との当たり判定処理を行います。

**Fig. 5-10　自機と弾との当たり判定処理**

## クラス構成

自機と弾との当たり判定処理を行うにあたって、Fig. 5-11のようにクラス構成を変更しました。自機に関しては、Chapter 4で作成した自機クラス（CMyShip）を基底クラスとして、3つの派生クラスを新規に作成します。また、自機の爆発を表現するために、2つの新しいクラスを追加し、ゲーム本体クラス（CShtGame）にも変更を加えます。

各クラスの役割は以下のとおりです。

Fig. 5-11 当たり判定処理を行うプログラムのクラス構成

### CMyShipクラス

Chapter 4で作成した自機のクラス（CMyShip）です。自機の共通機能をまとめています。このクラスから、自機のさまざまな状態を表すクラスを派生させます。

### CNormalMyShipクラス

通常時の自機を表すクラスです。弾との当たり判定処理を行います。弾に接触すると破壊されます。

### CAppearingMyShipクラス

出現時の自機を表すクラスです。ゲーム開始時に使います。弾との当たり判定処理は行わないため、弾に接触しても破壊されません。

### CRevivalMyShipクラス

復活時の自機を表すクラスです。自機が破壊された後に、再び自機を出現させるために使います。弾との当たり判定処理は行わないため、弾に接触しても破壊されません。出現時と復活時で自機のふるまいを変えられるように、CAppearingMyShipとは別のクラスとしました。

### CEffectクラス

爆発などの効果を表すクラスです。さまざまな効果の基底クラスとなります。

### CMyShipCrashクラス

自機の爆発を表すクラスです。自機が破壊されたときには、この爆発クラスのインスタンスを生成します。

### CShtGameクラス

ゲーム本体のクラスです。効果の移動処理などを追加します。

## 当たり判定処理

Fig. 5-12は、自機と弾の当たり判定処理の手順です。すべての弾についてループし、自機と弾の当たり判定が重なっているどうかを調べます。もしも重なっていたら、自機の耐久度を減らします。

当たり判定処理は、自機と弾、自機と敵といった、特定のキャラクター同士についてだけ行います。自機・弾・敵・効果といったキャラクターの種類に応じてタスクを分類し、それぞれ別々のタスクリストに格納しているのは、特定のタスク間でだけ当たり判定処理を行うためです(P. 73)。

自機は、弾や敵とだけ当たり判定処理を行うこととします。そのためにはFig. 5-13のように、キャラクターの種類に応じてタスクリストを分けておきます。そして、自機の当たり判定処理は、弾タスクリスト(BulletList)や敵タスクリスト(EnemyList)に属するタスクとの間でだけ行うようにします。

同様に、ショットと敵の当たり判定処理や、ビームと敵の当たり判定処理も行うことができます。それぞれ、ショットタスクリスト(ShotList)と敵タスクリストに属するタスク

**Fig. 5-12** 当たり判定処理の手順

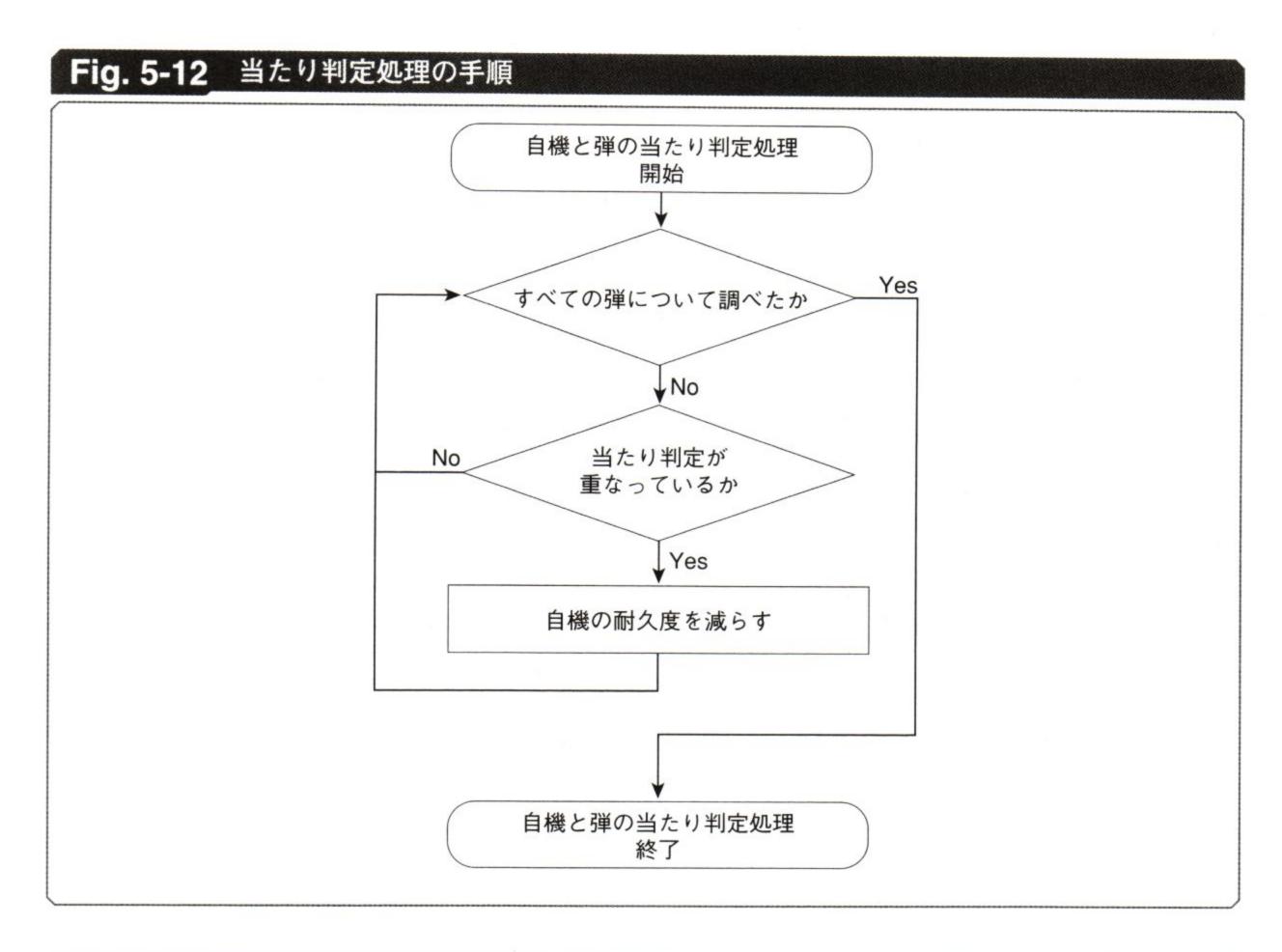



**Fig. 5-13** タスクの種類に応じた当たり判定処理

テキスト: TextList
（当たり判定処理を行わない）
100pts
ショット：ShotList
（当たり判定処理を行わない）
敵：EnemyList
（当たり判定処理を行う）
ビーム：BeamList
（当たり判定処理を行わない）
効果：EffectList
（当たり判定処理を行わない）
弾：BulletList
（当たり判定処理を行う）
自機：MyShipList
（当たり判定処理を行わない）
自機と当たり判定を
行わないタスク
自機と当たり判定を
行うタスク

の間、ビームタスクリスト (BeamList) と敵タスクリストに属するタスクの間で、当たり判定処理を行えばよいのです。

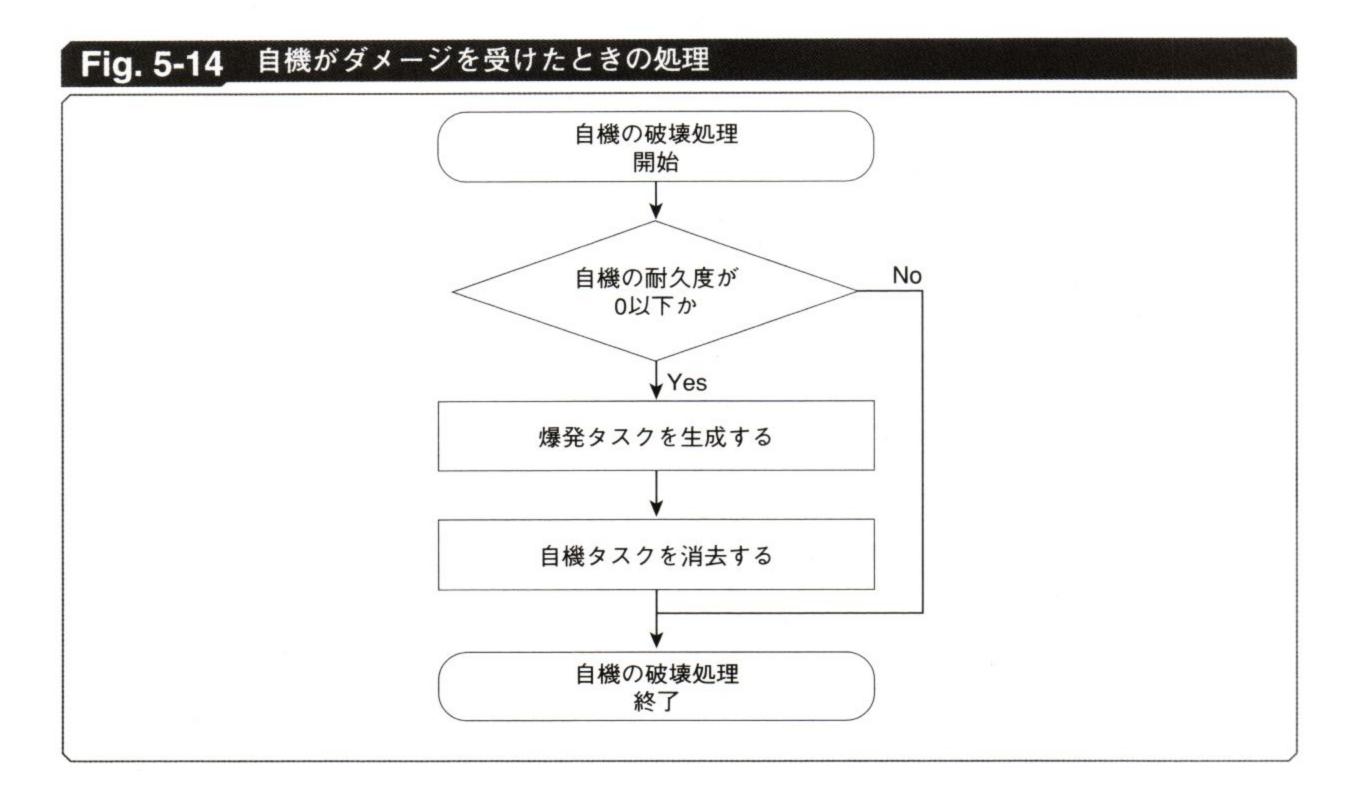
## 自機を破壊する

弾に接触した自機は破壊されます。1発の弾で自機が破壊されるゲームが多いのですが、自機が一定以上のダメージを受けるまで持ちこたえるゲームもあります。いずれにしても、基本的な処理は同じです。

Fig. 5-14は、自機の破壊を行うための手順です。まず、自機が破壊されたかどうかは、自機の耐久力から判定することができます。ここでは、被弾するたびに自機の耐久力を1ずつ減らしていき、0以下になったら破壊されることにします。自機の耐久力の初期値を1にしておけば、一度被弾しただけで自機が破壊されるようになります。また、当たった弾や敵の種類に応じてダメージ値が異なるというアレンジも可能です。

自機が破壊されたら、爆発や破片などのアニメーションを表示するのが一般的です。ここでは爆発のタスクを生成します。自機と同じ位置に爆発タスクを生成し、自機タスクを消去することによって、自機が爆発したように見せるわけです。

さらに、爆発タスクは一定時間が経過したら、残機がある場合にかぎり新しい自機を登場させます。そして自分 (爆発) を削除します。

**Fig. 5-14** 自機がダメージを受けたときの処理

# 当たり判定処理を行う自機のプログラム

当たり判定処理と破壊の処理は、自機の移動処理内で行います。List 5-5は、通常時の自機を表すクラス（CNormalMyShip）です。

耐久力が0以下になって自機が破壊されたら、自機の爆発を表すクラス（CMyShipCrash）のインスタンスを生成します。また、移動処理（Move関数）には、戻り値としてfalseを返させます。すると、自機タスクはゲーム本体によって消去されます。

自機が破壊されたときには、発射中のビームを消去する処理も行います。自機の破壊に合わせてビームを消去しないと、ビームだけが画面に残ってしまい不自然だからです。

自機の爆発時に呼び出しているSetMyShip関数は、ゲーム本体クラス（CShtGame）の関数です。この関数は、ゲーム本体クラスが保持する自機インスタンスへのポインタを設定します。このポインタは、自機に向かって狙い撃ち弾を発射するためなどに使います。

**List 5-5　通常時の自機を表すクラス（MyShip.h、MyShip.cpp）**

```
// 通常時の自機を表すクラス
class CNormalMyShip : public CMyShip {
protected:

    // 耐久力
    float Vit;

public:

    // コンストラクタ、移動
    CNormalMyShip(float x, float y);
    virtual bool Move();

};

// コンストラクタ
CNormalMyShip::CNormalMyShip(float x, float y)
:   CMyShip(x, y), Vit(1)
{}

// 移動
bool CNormalMyShip::Move() {

    // 自機に共通の移動処理
    CMyShip::Move();
```

```
    // 当たり判定処理（自機と弾）
    // Chapter 3のタスクイテレータクラス（CTaskIter）を使って
    // すべての弾タスクに関してループする
    for (CTaskIter i(Game->BulletList); i.HasNext(); ) {
        CBullet* bullet=(CBullet*)i.Next();

        // 接触の判定には移動物体クラス（CMover）の
        // Hit関数（List 5-1）を使う
        if (Hit(bullet)) {

            // 弾に接触したら耐久力を1だけ減らす
            Vit--;
        }
    }

    // 爆発
    if (Vit<=0) {
        new CMyShipCrash(X, Y);
        DeleteBeam();
        Game->SetMyShip(NULL);
        return false;
    }
    return true;
}
```

## 爆発のクラス

自機の爆発も、自機や弾などと同じくタスクとして表現します。本書では、爆発のような効果（エフェクト）の共通機能をまとめた効果クラス（CEffect）と、自機の爆発を表すクラス（CMyShipCrash）を定義します。

効果クラスは、自機クラス（CMyShip）や弾クラス（CBullet、List 5-2）と同様に、移動物体クラス（CMover）から派生させます。自機爆発クラスは、効果クラスから派生させます。

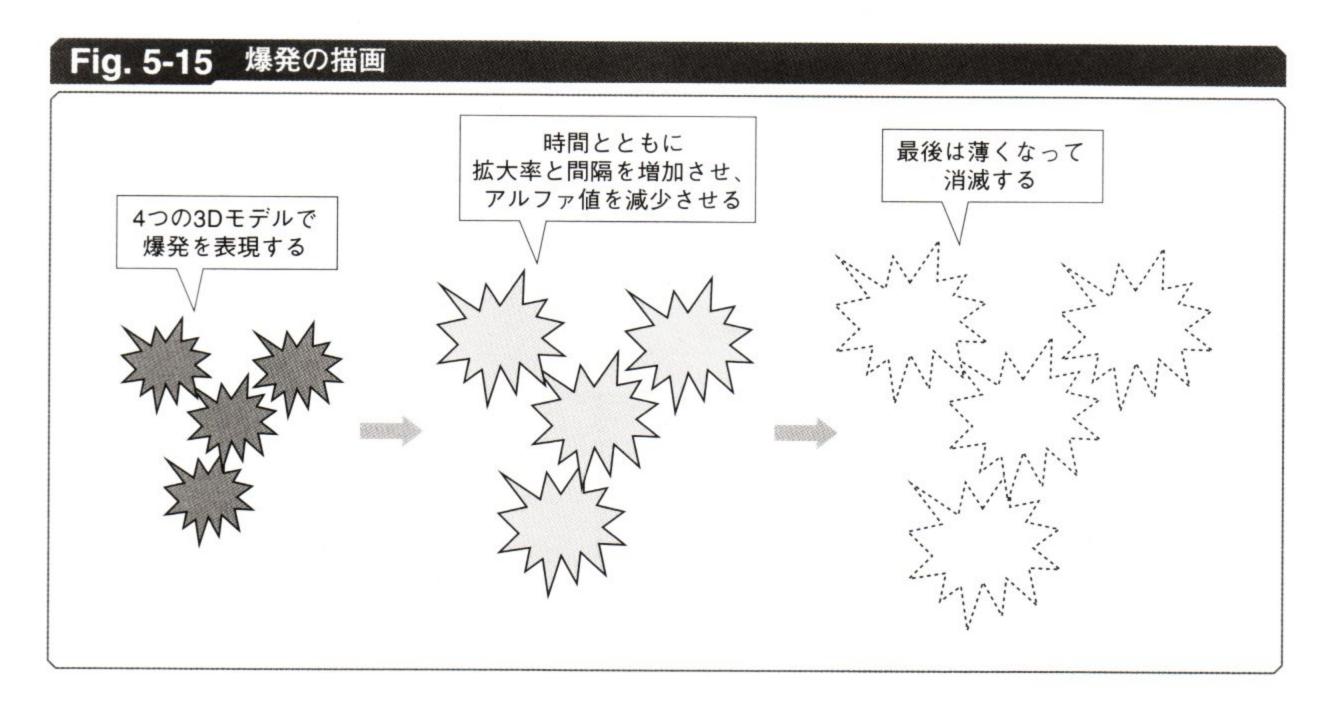
爆発を生成したら、まず爆発音を再生します。そして一定時間（サンプルでは50フレーム）が経過したら、自機を復活させ、自分（爆発）を消去します。

爆発の描画に関しては、Fig. 5-15のように4つの3Dモデルを表示しています。拡大率、アルファ値、位置を時間とともに少しずつ変化させることによって、広がりながら消えていく爆発を表現しました。

本書のサンプルは、ほとんどのキャラクターを1つの3Dモデルで表現していますが、描画処理は自由に記述することができるため、この爆発のように複数の3Dモデルを使って

キャラクターを描画することも可能です。また、3Dモデル以外の方法、例えば2D画像などを用いてキャラクターを描画することもできます。

List 5-6は、自機の爆発に関するプログラムです。

**Fig. 5-15 爆発の描画**

**List 5-6 爆発のクラス（Effect.h、Effect.cpp）**

```
// 効果のクラス
class CEffect : public CMover {
public:

    // タスクリストを指定するために
    // new演算子とdelete演算子をオーバーロードする
    // タスクリストは効果タスクリスト（EffectList）を使用
    void* operator new(size_t t) {
        return operator_new(t, Game->EffectList);
    }
    void operator delete(void* p) {
        operator_delete(p, Game->EffectList);
    }

    // コンストラクタ
    // 効果に当たり判定は必要ないので
    // 当たり判定を指定しない移動物体クラスの
    // コンストラクタ（List 5-1）を使用
    CEffect::CEffect(float x, float y)
    :   CMover(Game->EffectList, x, y, EFFECT_Z)
    {}
```

```
};

// 爆発のクラス
class CMyShipCrash : public CEffect {
protected:

    // タイマー
    int Time;

public:

    // コンストラクタ、移動、描画
    CMyShipCrash(float x, float y);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CMyShipCrash::CMyShipCrash(float x, float y)
:   CEffect(x, y), Time(0)
{
    // 爆発音を再生
    Game->SEBossCrash->Play();
}

// 移動
// 移動物体クラスの移動処理（Move関数）をオーバーライド
bool CMyShipCrash::Move() {
    Time++;
    if (Time>50) {

        // 新しい自機を復活させる
        Game->SetMyShip(new CRevivalMyShip(X, Y));

        // 爆発を消す
        return false;
    }
    return true;
}

// 描画
// 移動物体クラスの描画処理（Draw関数）をオーバーライド
void CMyShipCrash::Draw() {
    float
        scale=0.2f+0.2f*Time,
        alpha=1.0f-0.02f*Time,
```

```
        dist=0.5f*Time;
    Game->MeshCrashSauce->Draw(
        X, Z, -Y,
        scale, 1, scale, 0, 0, 0, TO_ZYX, alpha, false);
    Game->MeshCrashSauce->Draw(
        X+dist*cos(D3DX_PI*0.12f), Z, -Y+dist*sin(D3DX_PI*0.12f),
        scale, 1, scale, 0, 0, 0, TO_ZYX, alpha, false);
    Game->MeshCrashSauce->Draw(
        X+dist*cos(D3DX_PI*0.78f), Z, -Y+dist*sin(D3DX_PI*0.78f),
        scale, 1, scale, 0, 0, 0, TO_ZYX, alpha, false);
    Game->MeshCrashSauce->Draw(
        X+dist*cos(D3DX_PI*1.44f), Z, -Y+dist*sin(D3DX_PI*1.44f),
        scale, 1, scale, 0, 0, 0, TO_ZYX, alpha, false);
}
```

## 復活時の自機

弾や敵などに接触すると自機は破壊されて、ミスとなります。ミスの後に再び自機を復活させる際には、ある程度の無敵期間を設けると遊びやすくなります。出現した直後にまた自機が破壊されてしまうと、プレイヤーはおそらくストレスを感じてしまうからです。

そこで、通常の自機を表すクラス（CNormalMyShip）とは別に、復活時の自機を表すクラス（CRevivalMyShip）を定義しました。復活時の自機は、通常の自機と同様に上下左右へ動かすことができます。しかし、当たり判定処理を行わないので、ダメージを受けることはありません。

また、復活時の自機はタイマーを保持していて、一定時間が経過したら通常の自機を生成します。このとき、自分（復活時の自機）を消去することにより、復活時の自機から通常時の自機に状態が変化したように見せます。通常の自機の出現位置は、復活時の自機の消滅位置と同じにしておきます。

復活時の自機は、描画の方法も通常の自機とは変えておいた方がよいでしょう。無敵時間中であることをわかりやすく示すためです。サンプルでは、自機に点滅するエフェクトを重ねて描画しています。

List 5-7は、復活時の自機を表すクラスです。移動処理（Move関数）では、自機の共通クラス（CMyShip）の移動処理を呼び出すことによって、通常の自機と同様に上下左右へ動かせるようにしています。

**List 5-7 復活時の自機（MyShip.h、MyShip.cpp）**

```
// 復活時の自機（点滅していて当たり判定がない）
class CRevivalMyShip : public CMyShip {
protected:

    // タイマー、
    int Time;

public:

    // コンストラクタ、移動、描画
    CRevivalMyShip(float x, float y);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CRevivalMyShip::CRevivalMyShip(float x, float y)
:   CMyShip(x, y), Time(0)
{}

// 移動
bool CRevivalMyShip::Move() {

    // 共通の移動処理
    CMyShip::Move();

    // 自機を点滅させる
    Alpha=0.5f+0.5f*(float)sin(Time*0.7f);

    // 一定時間が経過したら通常の自機に変わる
    //（通常の自機を生成して自分は消える）
    Time++;
    if (Time>120) {
        new CNormalMyShip(X, Y);
        return false;
    }
    return true;
}

// 描画
void CRevivalMyShip::Draw() {

    // エフェクトの描画
    float
        scale=3.5f+0.5f*sin(Time*0.5f),
```

```
        turn=(float)Time/10;
    Game->MeshCrashSauce->Draw(
        X, Z, -Y, scale, 1, scale,
        0, turn, 0, TO_Y, Alpha, false);

    // 自機の描画
    CMyShip::Draw();
}
```

## 出現時の自機

ゲーム開始時に表示される出現時の自機も、通常の自機とは違った動きをします。そこで、出現時の自機を表すクラス（CAppearingMyShip）を定義します。

出現時の自機は画面下部の画面枠外から登場し、画面中央やや下まで移動します。自機が画面中央やや下に達したら、通常の自機に変化させます。この変化は、出現時の自機を消去し、通常の自機を生成することによって実現します。

List 5-8は、出現時の自機を表すクラスです。このクラスは、自機の共通クラス（CMyShip）から派生しています。描画は共通の処理を使い、移動処理（Move関数）だけをオーバーライドします。

**List 5-8　出現時の自機（MyShip.h、MyShip.cpp）**

```
// 出現時の自機（画面の下から出てくる、操作はできない）
class CAppearingMyShip : public CMyShip {
public:

    // コンストラクタ、移動、描画
    CAppearingMyShip();
    virtual bool Move();
};

// コンストラクタ
CAppearingMyShip::CAppearingMyShip()
:   CMyShip(0, 60)
{}

// 移動
bool CAppearingMyShip::Move() {

    // 画面の下から出てくる
    Y-=0.5f;
```

```
    // ロールする
    Roll+=0.1f;

    // 目標の位置に達したら、
    // 通常の自機を生成し、自分は消える
    if (Y<=30) {
        new CNormalMyShip(X, Y);
        return false;
    }
    return true;
}
```

## いろいろな弾

さて、ゲーム画面に弾が登場するようになりました。また、弾と自機との当たり判定処理や、自機を破壊する処理もできました。これで、かなりシューティングゲームらしくなってきました。しかし、基本的な方向弾を作っただけでは物足りないので、もう少しいろいろな弾を作ってみたいと思います。

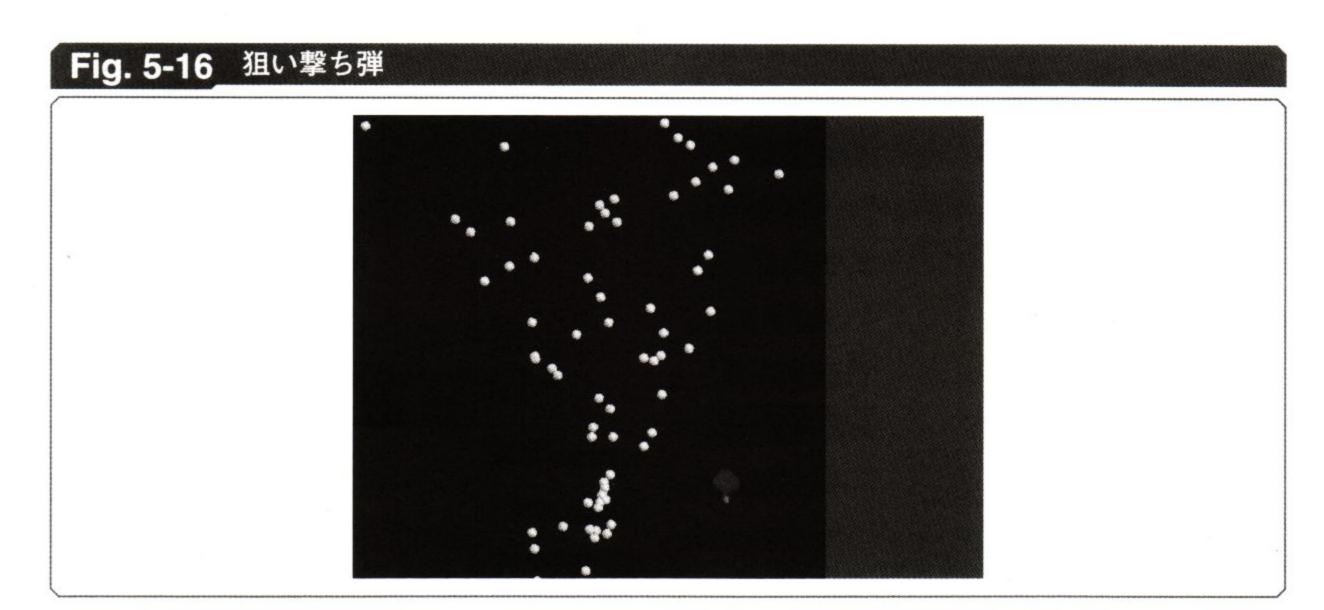
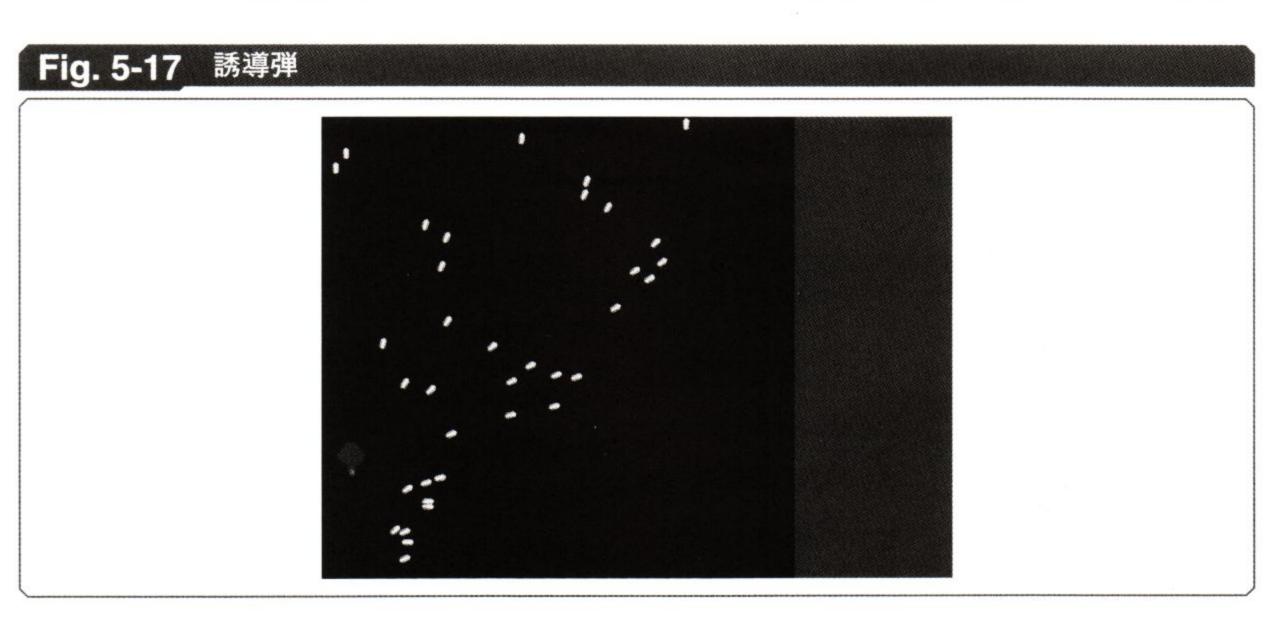
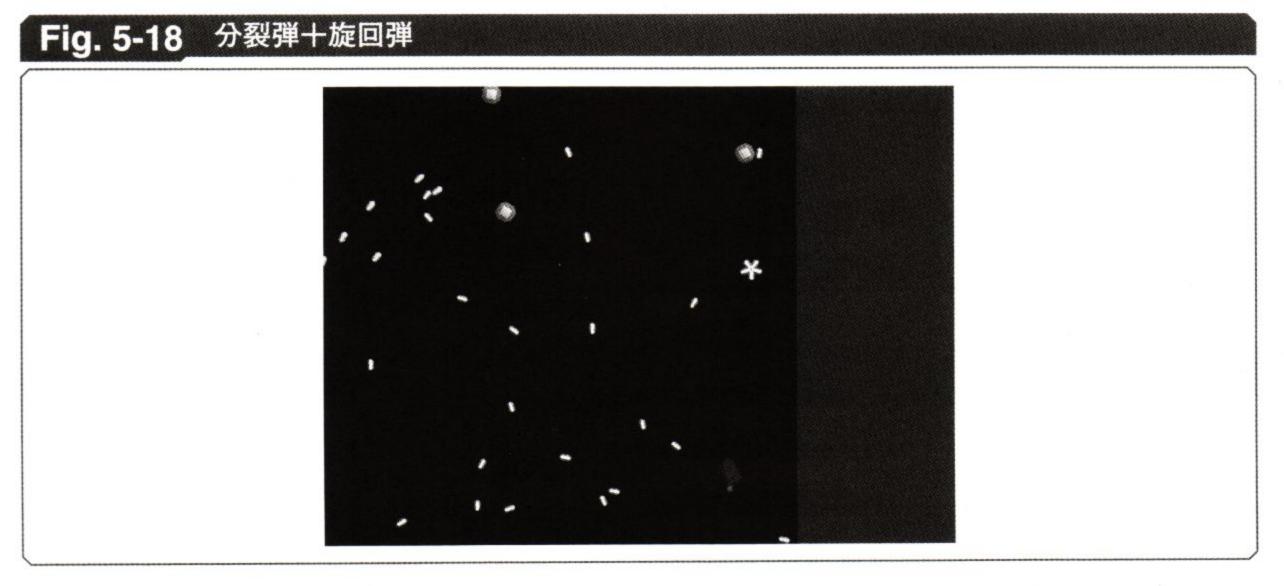
ここで紹介するのは「狙い撃ち弾」「誘導弾」「分裂弾」「旋回弾」です。この他のさまざまな弾のアルゴリズムについては書籍『シューティングゲームアルゴリズムマニアックス』に記載していますので、あわせてご利用いただければ幸いです。

Fig. 5-16～18はいろいろな弾を追加したサンプルの実行画面です。プロジェクトは付録CD-ROMの「ShtGame_Bullet2」フォルダに収録しています。実行ファイルは「ShtGame_Bullet2¥Release¥ShtGame.exe」です。

サンプルでは、約5秒ごとに弾の種類が変わります。弾に接触すると自機は破壊されますが、少し時間がたつと復活します。自機を操作して弾を避けるだけですが、弾の種類が増えたので展開に変化が感じられるようになったと思います。まだ短いパターンが繰り返されるだけですが、十分長いパターンに発展させれば、弾避け専門のゲームに仕上げることも可能でしょう。

Fig. 5-16 狙い撃ち弾

Fig. 5-17 誘導弾

Fig. 5-18 分裂弾＋旋回弾

## 狙い撃ち弾

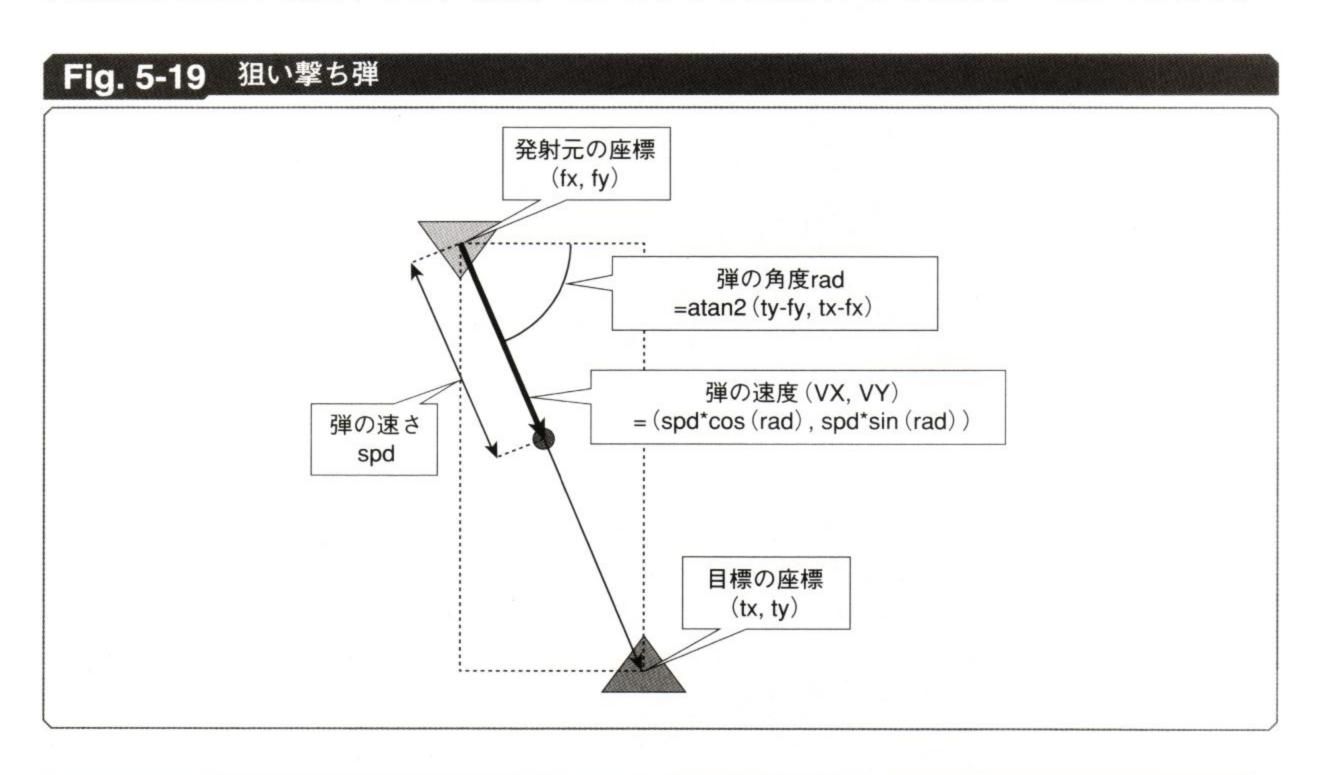
ここでは目標に向かってまっすぐ飛ぶ弾を「狙い撃ち弾」と呼ぶことにします。狙い撃ち弾は、方向弾と並んで最も基本的な弾の一種です。狙い撃ち弾は、発射後に目標の位置が変わっても、進路を変更することはしません。一方、のちほど紹介する誘導弾は進路を修正します（P. 174）。

狙い撃ち弾の処理は、狙い撃ち弾のクラス（CAimBullet）にまとめました。このクラスは、方向弾のクラス（CDirBullet、List 5-3）と構成がよく似ています。

狙い撃ち弾のポイントは生成処理です。発射元の座標から、目標の座標に向けて、一定の速さで飛ぶように弾の速度を設定します。狙い撃ち弾の速度計算についてはFig. 5-19にまとめました。

速度とともに、加速度と回転角度も計算します。回転角度は、弾の形状として針状の3Dモデルを使ったときに、針の先端が目標を向くように調整しています。

List 5-9は、狙い撃ち弾を表すCAimBulletクラスのプログラムです。コンストラクタで引数dirを0以外に設定すると、目標から少しずらした方向に弾を飛ばすことができます。

**Fig. 5-19　狙い撃ち弾**

**List 5-9　狙い撃ち弾（Bullet.h、Bullet.cpp）**

```
// 狙い撃ち弾のクラス
class CAimBullet : public CBullet {
```

```
    // 速度、加速度
    float VX, VY, AX, AY;

public:

    // コンストラクタ、移動
    CAimBullet(
        CMesh** mesh, int color,
        float fx, float fy, float tx, float ty,
        float dir, float spd, float accel);
    virtual bool Move();
};

// コンストラクタ
// 引数の内容は以下のとおり
// 発射元の座標（fx, fy）
// 目標の座標（tx, ty）
// 速さspd
// 加速の大きさaccel
// 目標からのずれdir
CAimBullet::CAimBullet(
    CMesh** mesh, int color,
    float fx, float fy, float tx, float ty,
    float dir, float spd, float accel
)
:   CBullet(mesh, color, fx, fy)
{
    // 発射元（fx, fy）から目標（tx, ty）に向けて
    // 速さspdで飛ぶように速度を設定する
    // ただし、角度dirだけ針路をずらす
    float
        rad=atan2(ty-fy, tx-fx)+D3DX_PI*2*dir,
        c=cos(rad), s=sin(rad);
    VX=spd*c;
    VY=spd*s;

    // 加速度
    AX=accel*c;
    AY=accel*s;

    // 角度
    Yaw=rad*(0.5f/D3DX_PI)+0.25f;
}

// 移動
bool CAimBullet::Move() {
```

```
    // 座標の更新
    X+=VX;
    Y+=VY;

    // 速度の更新
    VX+=AX;
    VY+=AY;

    // 共通処理（弾を回転させ、画面から出たときに消去する）
    return CBullet::Move();
}
```

## 誘導弾

誘導弾は自機に向かってしだいに近づいてくる弾です。自機の方向を調べて、弾の進行方向を修正することにより、自機へ向かって誘導します (Fig. 5-20)。

このとき、誘導の性能が高すぎると、自機が弾を避けることができなくなってしまいます。適度に避けられる誘導弾にするには、旋回角度に上限を設ける方法などがあります (Fig. 5-21)。

自機の方向が旋回可能範囲内の場合には、弾の進行方向を自機の方向にします (Fig. 5-22)。旋回可能範囲外の場合には、自機の方向に近づくよう、旋回角度の上限だけ進行方向を曲げます (Fig. 5-23)。

これをプログラムにしたのが、List 5-10の誘導弾クラス (CHomingBullet) です。狙い撃ち弾クラス (List 5-9) と同様に、クラス定義、コンストラクタ、移動処理 (Move関数) から構成されています。

**Fig. 5-20 誘導弾**

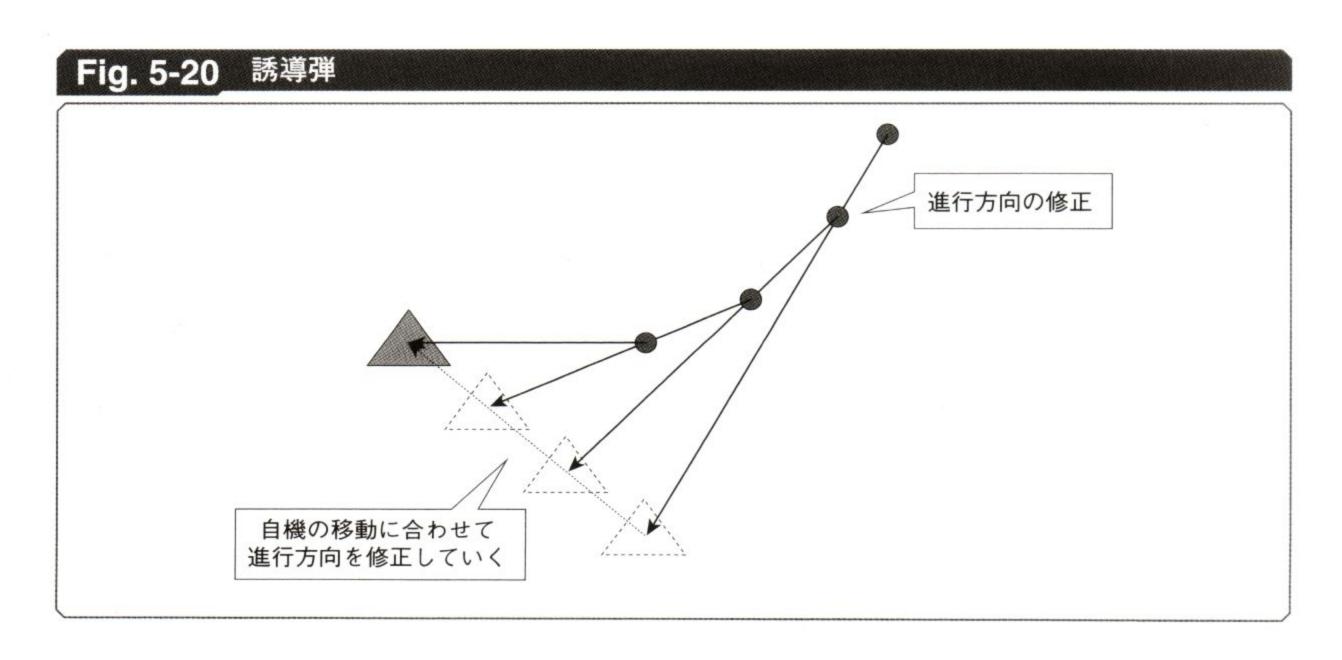



**Fig. 5-21　旋回角度に上限を設ける**

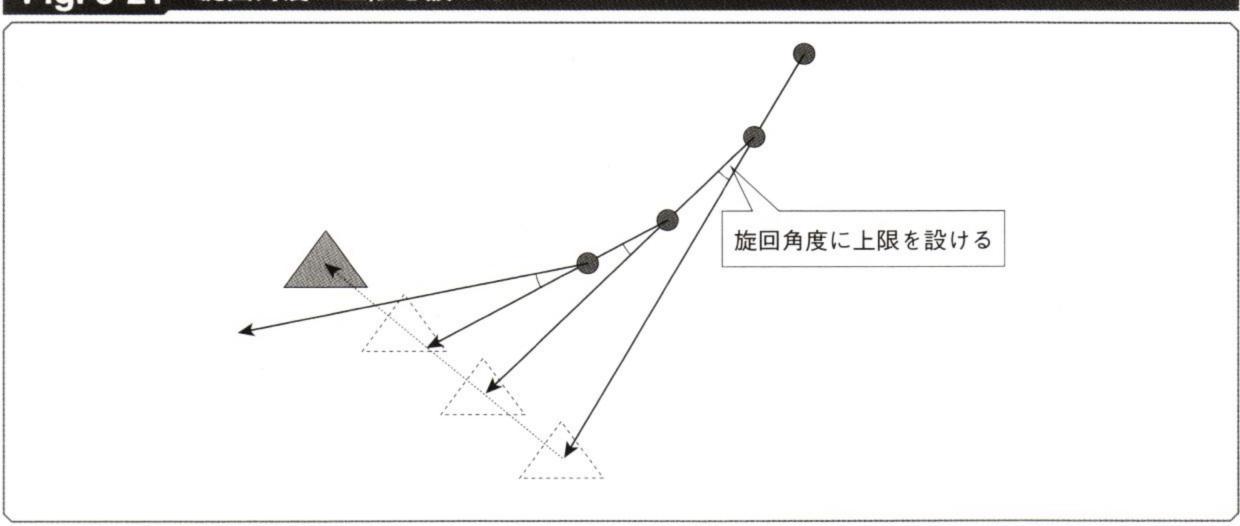



**Fig. 5-22　旋回可能範囲内の場合の動き**

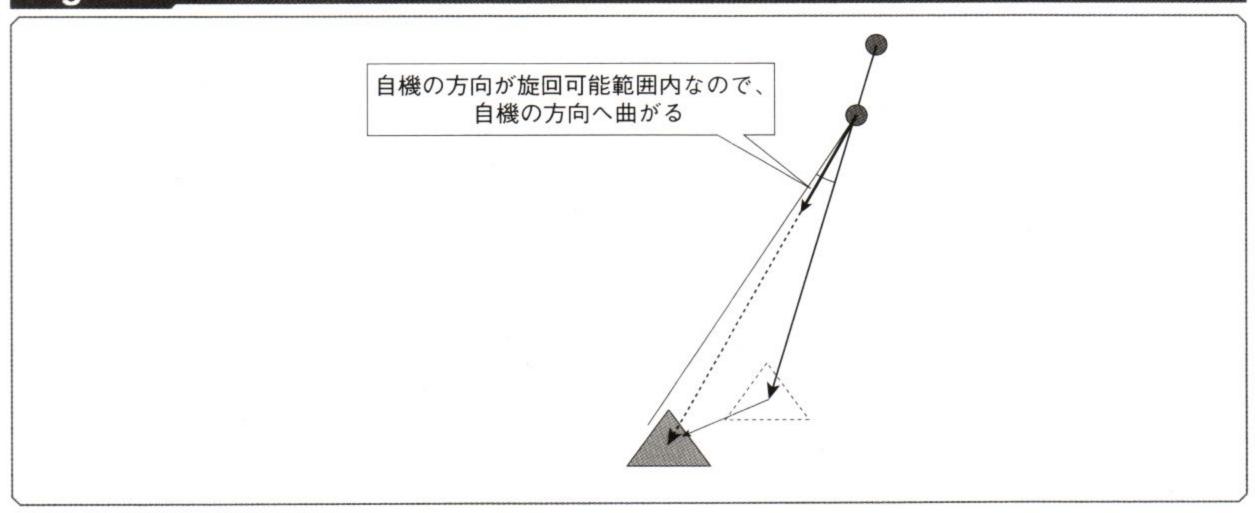
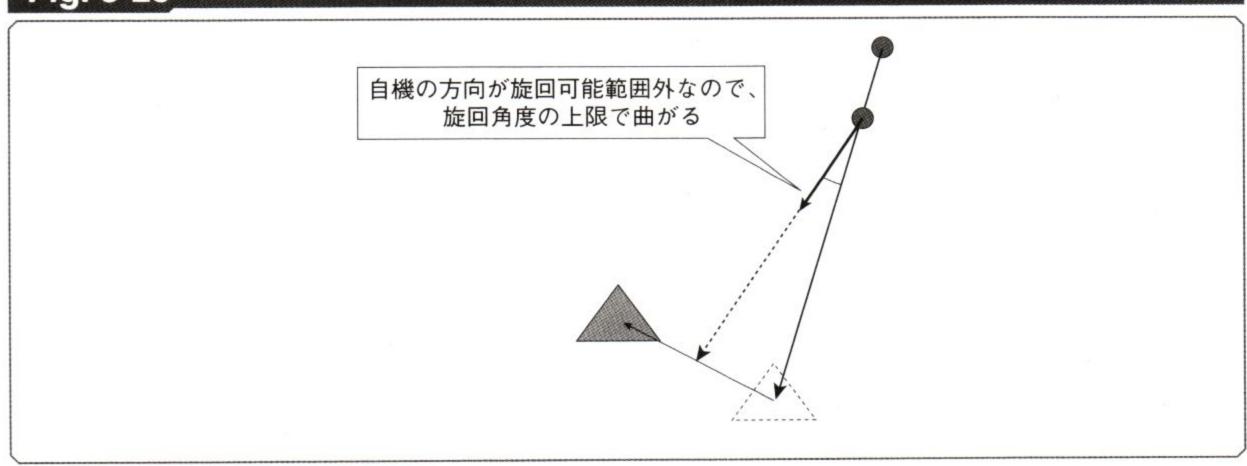



**Fig. 5-23　旋回可能範囲外の場合の動き**

**List 5-10 誘導弾（Bullet.h、Bullet.cpp）**

```
// 誘導弾のクラス
class CHomingBullet : public CBullet {

    // 進行方向、進行方向の変化、移動の速さ
    float Rad, VRad, Speed;

public:

    // コンストラクタ、移動
    CHomingBullet(
        CMesh** mesh, int color, float x, float y,
        float rad, float vrad, float spd);
    virtual bool Move();
};

// コンストラクタ
CHomingBullet::CHomingBullet(
    CMesh** mesh, int color, float x, float y,
    float rad, float vrad, float spd
)
:   CBullet(mesh, color, x, y), Rad(rad), VRad(vrad), Speed(spd)
{}

// 移動
bool CHomingBullet::Move() {

    // 自機が画面上に存在するときには誘導する
    CMyShip* myship=Game->GetMyShip();
    if (myship) {

        // 自機の方向を求める
        float newrad=atan2(myship->Y-Y, myship->X-X);

        // 自機方向と進行方向の差が一定範囲（VRad）内ならば、
        // 弾を自機に向ける
        if (abs(newrad-Rad)<VRad) {
            Rad=newrad;
        }

        // 自機方向と進行方向の差が一定範囲（VRad）外ならば、
        // 進行方向を自機方向にVradだけ近づける
        else {
            if (newrad<Rad-D3DX_PI) Rad-=D3DX_PI*2; else
            if (newrad>Rad+D3DX_PI) Rad+=D3DX_PI*2;
            if (newrad<Rad) Rad-=VRad; else Rad+=VRad;
```

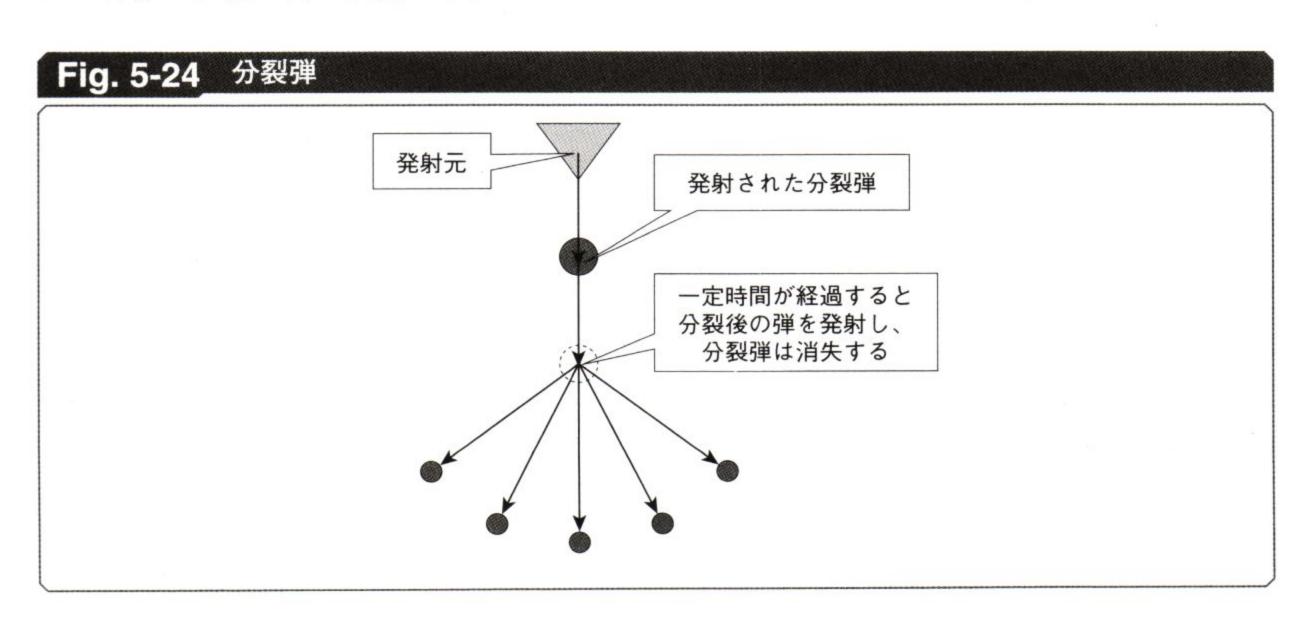
```
        }
    }

    // 座標の更新
    X+=cos(Rad)*Speed;
    Y+=sin(Rad)*Speed;

    // 弾を回転させない場合は、弾を進行方向に向ける
    if (VYaw==0) Yaw=Rad*(0.5f/D3DX_PI)+0.25f;

    return CBullet::Move();
}
```

## 分裂弾

次は弾を分裂させてみましょう。弾を分裂させるような処理は、タスクシステムをベースにしたプログラムの得意とするところです。

Fig. 5-24が分裂弾の仕組みです。弾を分裂させるには、一定時間が経過した後に、分裂前の弾の位置から分裂後の弾を発射します。これは新しい弾のタスクを生成することに相当します。そして、元の分裂弾のタスクは消去します。これで、弾が分裂したような効果を得ることができます。サンプルでは、次に説明する旋回弾を扇状に生成していますが、別の種類の弾を生成することも可能です。

分裂する前の分裂弾は、他の弾よりも少し大きめに描画することにしました。これは、プレイヤーに警戒をうながすためです。弾を回転させたり、明滅させたり、演出方法はいろいろと考えられます。サンプルでは、3Dモデルを拡大して描画しています。弾の3Dモデル自体は、他の弾と共通です。

**Fig. 5-24** 分裂弾

List 5-11は、分裂弾クラス（CSplitBullet）です。構成は狙い撃ち弾や誘導弾とほぼ同じになっています。

**List 5-11　分裂弾（Bullet.h、Bullet.cpp）**

```
// 分裂弾のクラス
class CSplitBullet : public CBullet {

    // 速度、方向、速さ、タイマー
    float VX, VY, Dir, Spd;
    int Time;

public:

    // コンストラクタ、移動、描画
    CSplitBullet(
        CMesh** mesh, int color, float x, float y,
        float dir, float spd);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CSplitBullet::CSplitBullet(
    CMesh** mesh, int color, float x, float y, float dir, float spd
)
:   CBullet(mesh, color, x, y), Dir(dir), Spd(spd), Time(50)
{
    // 速度と回転角度の設定
    VX=spd*cos(D3DX_PI*2*dir);
    VY=spd*sin(D3DX_PI*2*dir);
    Yaw=Dir+0.25f;
}

// 移動
bool CSplitBullet::Move() {

    // 座標の更新
    X+=VX;
    Y+=VY;

    // 一定時間が経過したら分裂する
    Time--;
    if (Time==0) {

        // 分裂後の弾を発射する（タスクを生成する）
```

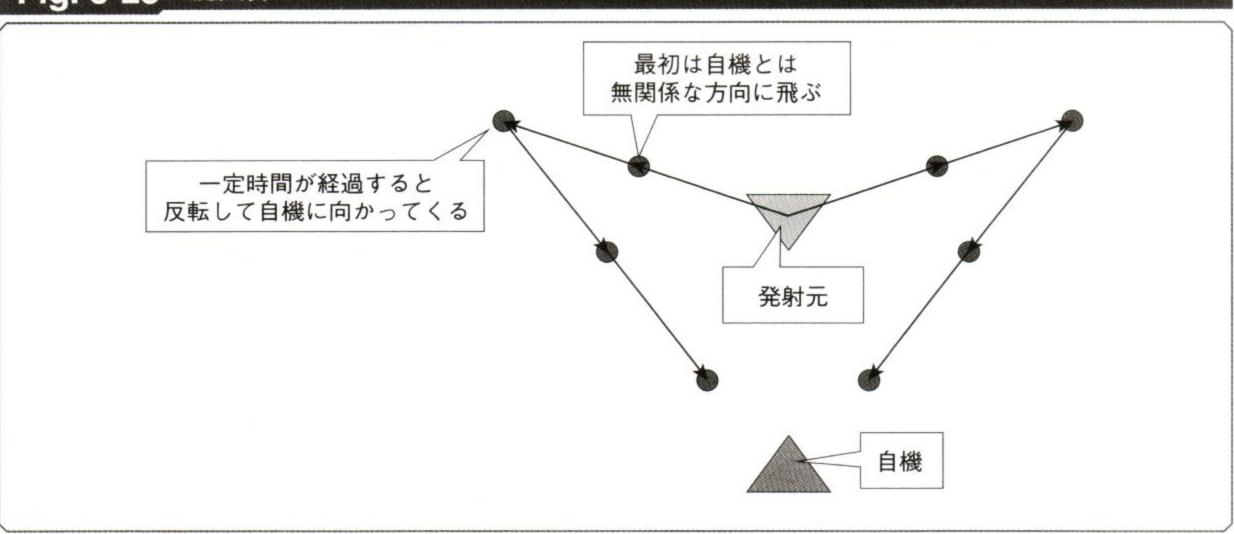
```
        float spd=Spd*1.5f, dir=Dir-0.4f;
        for (int i=0; i<5; i++, dir+=0.2f) {

            // 旋回弾を生成
            // 他の種類の弾を生成することも可能
            new CTurnBullet(
                Game->MeshNeedle, Color, X, Y, dir, spd);
        }

        // 自分（分裂弾）は消去する
        return false;
    }
    return CBullet::Move();
}

// 描画
void CSplitBullet::Draw() {
    Mesh->Draw(X, Z, -Y, 2, 2, 2, 0, Yaw, 0, TO_ZYX, 1, false);
}
```

## 旋回弾

弾の動きに変化をつけるには、途中で性質が変わる弾を使う方法もあります。そこで、ここでは旋回弾と称して、方向弾が途中から狙い撃ち弾に変化するような処理を実現してみます。これはFig. 5-25のように、最初は自機とは無関係の方向に進み、途中から反転して自機に向かってくるような弾です。

旋回弾の処理は、分裂弾の処理に似ています。最初は方向弾として弾を飛ばして、一定時間が経過した後に、狙い撃ち弾として弾の速度を再計算します。こうすれば、途中から自機に向かってくるような効果を得ることができます。

旋回弾クラス（CTurnBullet）はList 5-12のようになります。ポイントは移動処理です。分裂弾と同じく、変数（Time）を使用して時間を管理し、一定時間が経過したら速度を再計算します。この再計算方法は、狙い撃ち弾（List 5-9）の計算方法と同じです。ただし分裂弾とは違い、同じタスクのまま動きを変化させるだけなので、タスクの消去は行いません。

**Fig. 5-25 旋回弾**

**List 5-12 旋回弾(Bullet.h、Bullet.cpp)**

```
// 旋回弾
class CTurnBullet : public CBullet {

    // 速度、方向、速さ、タイマー
    float VX, VY, Spd;
    int Time;

public:

    // コンストラクタ、移動、描画
    CTurnBullet(
        CMesh** mesh, int color, float x, float y,
        float dir, float spd);
    virtual bool Move();
};

// コンストラクタ
CTurnBullet::CTurnBullet(
    CMesh** mesh, int color, float x, float y,
    float dir, float spd
)
:   CBullet(mesh, color, x, y), Spd(spd), Time(30)
{
    VX=spd*cos(D3DX_PI*2*dir);
    VY=spd*sin(D3DX_PI*2*dir);
    Yaw=dir+0.25f;
}
```

```
// 移動
bool CTurnBullet::Move() {
    CMyShip* myship=Game->GetMyShip();

    // 座標の更新
    X+=VX;
    Y+=VY;

    // 一定時間が経過したら狙い撃ち弾に変化する
    Time--;
    if (Time==0 && myship) {

        // 自機に向かうように速度を再計算する
        float rad=atan2(myship->Y-Y, myship->X-X);
        VX=Spd*cos(rad);
        VY=Spd*sin(rad);
        Yaw=rad/(D3DX_PI*2)+0.25f;
    }
    return CBullet::Move();
}
```

## いろいろな弾の発射

ここまでに作成したいろいろな弾は、List 5-13のようなプログラムで発射しています。これはゲーム本体を表すクラス（CShtGame）の移動処理（Move関数）です。

ここではタイマーを使って、300フレーム（約5秒）ごとに弾の種類を変えながら、いろいろな弾を発射します。弾を発射するには、方向弾（CDirBullet）や狙い撃ち弾（CAimBullet）といった各種の弾のインスタンスを生成します。また、MoveTask関数（P. 112）を用いて、弾・ビーム・効果・自機・ショットのタスクをそれぞれ動かしています。

**List 5-13 いろいろな弾の発射（Main.cpp）**

```
void CShtGame::Move() {

    // ...（中略）...

    // 弾の生成
    CMyShip* myship=Game->GetMyShip();
    switch (Time/300) {

        // 方向弾
        case 0:
            if (Time%8==0) {
```

```
                float dir=Rand05()*0.06f;
                for (int i=-10; i<=10; i++) {
                    new CDirBullet(
                        Game->MeshNeedle, 0, 0, -50,
                        0.25f+dir+0.03f*i, 0.5f, 0.01f);
                }
            }
            break;

        // 狙い撃ち弾
        case 1:
            if (Time%2==0) {
                if (myship) {
                    new CAimBullet(
                        Game->MeshBullet, 1, Rand05()*100, -50,
                        myship->X, myship->Y, 0, 0.8f, 0);
                }
            }
            break;

        // 誘導弾
        case 2:
            if (Time%4==0) {
                new CHomingBullet(
                    Game->MeshNeedle, 1, Rand05()*100, -50,
                    0.25f*D3DX_PI*2, 0.02f, 0.8f);
            }
            break;

        // 分裂旋回弾
        case 3:
            if (Time%16==0) {
                new CSplitBullet(
                    Game->MeshBullet, 0, Rand05()*100, -50,
                    0.25f, 0.8f);
            }
            break;

        default:
            Time=0;
    }
    Time++;

    // タスクの動作
    MoveTask(BulletList);
    MoveTask(BeamList);
```

```
        MoveTask(EffectList);
        MoveTask(MyShipList);
        MoveTask(ShotList);
}
```

## 弾幕を作る

いわゆる弾幕系と称されるシューティングゲームのなかには、非常に複雑なパターンの弾幕を生成する作品が多く見られます。しかし落ち着いて観察してみると、実はこれらの弾幕は基本的な弾の動きの組み合わせからできていることがわかります。複雑な弾幕を構成する際のポイントの1つは、さまざまな弾の動きを組み合わせることです。

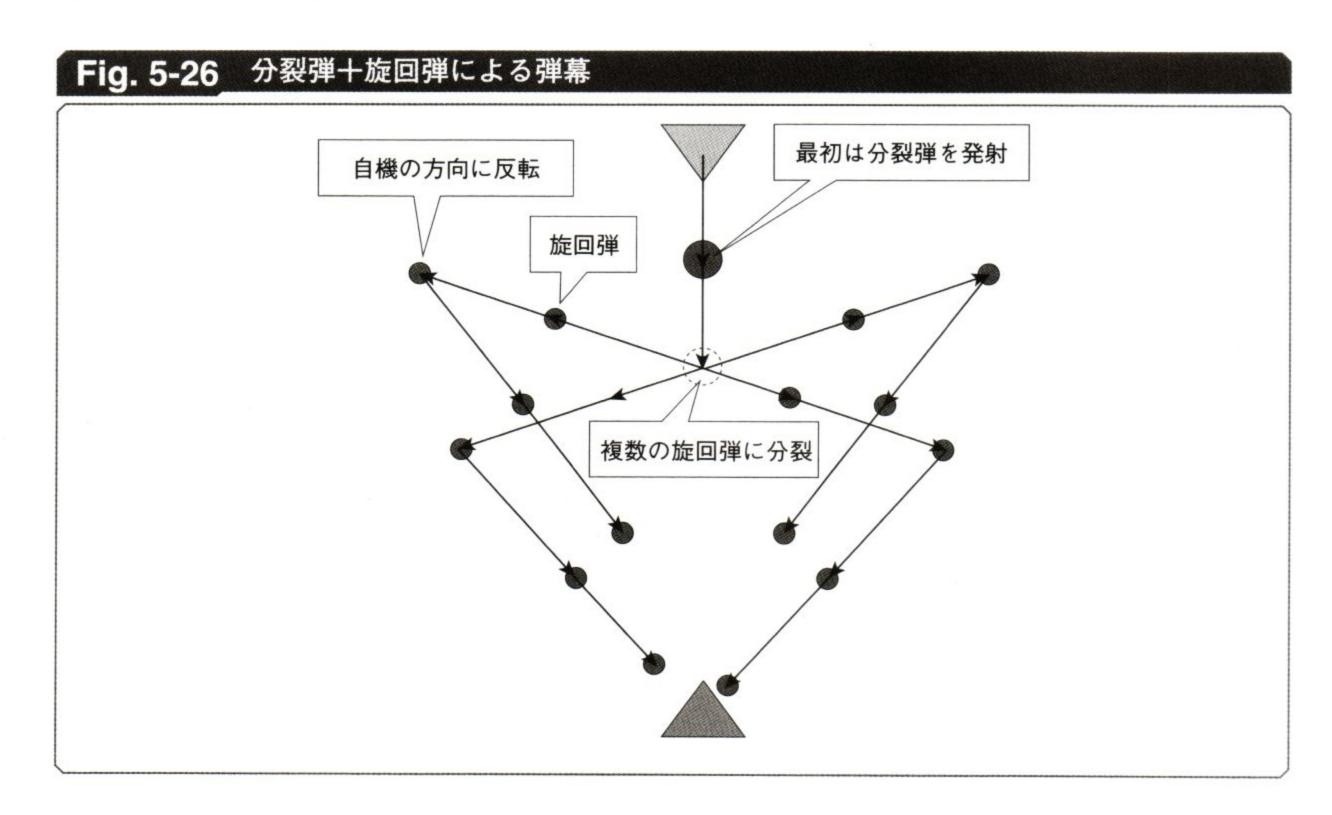
例えば、先ほど解説した分裂弾と旋回弾を組み合わせるだけで、Fig. 5-26のように複雑な動きをする弾幕を構成することができます。最初に敵から分裂弾が発射され、一定時間が経過すると複数の旋回弾に分裂します。これらの旋回弾はしばらく直進しますが、再び一定時間が経過すると、向きを変えて自機の方向に飛んできます。サンプルを実行して、実際の動きをぜひご覧ください。

弾幕を作るときのもう1つのポイントは、加速度を使うことです。加速度を使うと、だんだん速くなる弾や、逆にだんだん遅くなる弾を作ることができます。さらに、弾によって加速度を変えると、ある弾が別の弾を追い越したり、逆に追い越されたりといったことが起きます。こういった動きを使うと、非常に複雑で、見た目にも面白い弾幕を生成することができます。

多くのゲームでは、弾幕を何種類か組み合わせることによって、さらに複雑かつ画面全体をおおいつくすような弾幕を構成しています。個々の弾に関するプログラムは、ここで解説したように比較的簡単に作成することができます。あとは、以下の点が、魅力的なゲームを作り上げるためのカギとなるでしょう。

- 弾や弾幕のバリエーションをなるべく多く用意する
- 複数の弾幕を組み合わせて、見るのも避けるのも楽しい弾幕を構成する
- シーンごとに効果的な弾幕を選択する
- 多少は無駄な弾も撃つことによって画面を派手に見せる

また、性質の異なる弾を異なる色で表示すると、プレイヤーが弾筋を見分けやすくなり、画面も派手にできます。しかし、弾に使用する色の数が多すぎると、背景と弾、あるいは自機と弾などの見分けがつきにくくなってしまいます。作成するゲームの特徴を考慮して、必要最小限の数の色を使うのがお勧めです。

**Fig. 5-26** 分裂弾＋旋回弾による弾幕

## ＞＞Chapter 5のまとめ

本章では弾の移動と表示、自機と弾との当たり判定処理、自機の破壊、そして弾幕の作り方について解説しました。ここまでで、とりあえずゲームとして遊べるプログラムが作成できました。次章以降の要素は、好みやゲームバランスに応じて追加していただくとよいでしょう。

次章では敵をテーマに、敵の移動と表示、敵が弾を撃つ処理、ショットおよびレーザーと敵との当たり判定処理、得点の加算と表示などについて説明します。弾幕だけでは物足りない、という方は、ぜひ続けて敵の制作に取り組んでください。

# Chapter 06

本章では敵に関するプログラムを作ります。敵の移動・表示、弾を撃つ処理、ショットやビームとの当たり判定処理、スコアの加算・表示について説明します。
Chapter 1から本章までで、シューティングゲームの中核となる要素はできあがりです。自機・弾・敵の三者がそろえば、自作のプログラムで遊びながら、より楽しく開発を進められるでしょう。もう一息ですので頑張ってください。

# 敵の作り方

自機と弾のプログラムがある程度できあがったら、今度は敵を作ってみましょう。敵の作り方は、Chapter 5で解説した弾の作り方によく似ています。敵と弾は、破壊の可否やスコアの有無といった性質は違いますが、基本的にはよく似たものです。敵を作るには、弾の場合と同様に、敵のクラスを定義して、移動処理や描画処理を記述します。

Fig. 6-1は敵を追加したサンプルの実行画面です。プロジェクトは付録CD-ROMの「ShtGame_Enemy」フォルダに収録しました。実行ファイルは「ShtGame_Enemy¥Release¥ShtGame.exe」です。

このサンプルでは、画面上方から敵が出現します。敵には赤身・玉子・海老の3種類があり（いずれも寿司ネタです）、一定時間ごとに出現する敵の種類が変わります。

敵はショットまたはビームで破壊することができます。敵を破壊するとスコア（得点）が入ります。スコアは画面右側に表示されます。

敵は弾を発射します。弾や敵に接触すると自機は破壊されてしまいますが、少し時間がたつと復活します。

敵を作ると、ずいぶんとシューティングゲームらしくなってきます。本章では、最初に敵を動かす方法を説明し、次にショットやビームと敵の当たり判定処理と、敵を破壊する処理について解説します。最後に、スコアの加算や表示についても述べます。

**Fig. 6-1** 敵を追加したサンプル

# 敵のプログラム

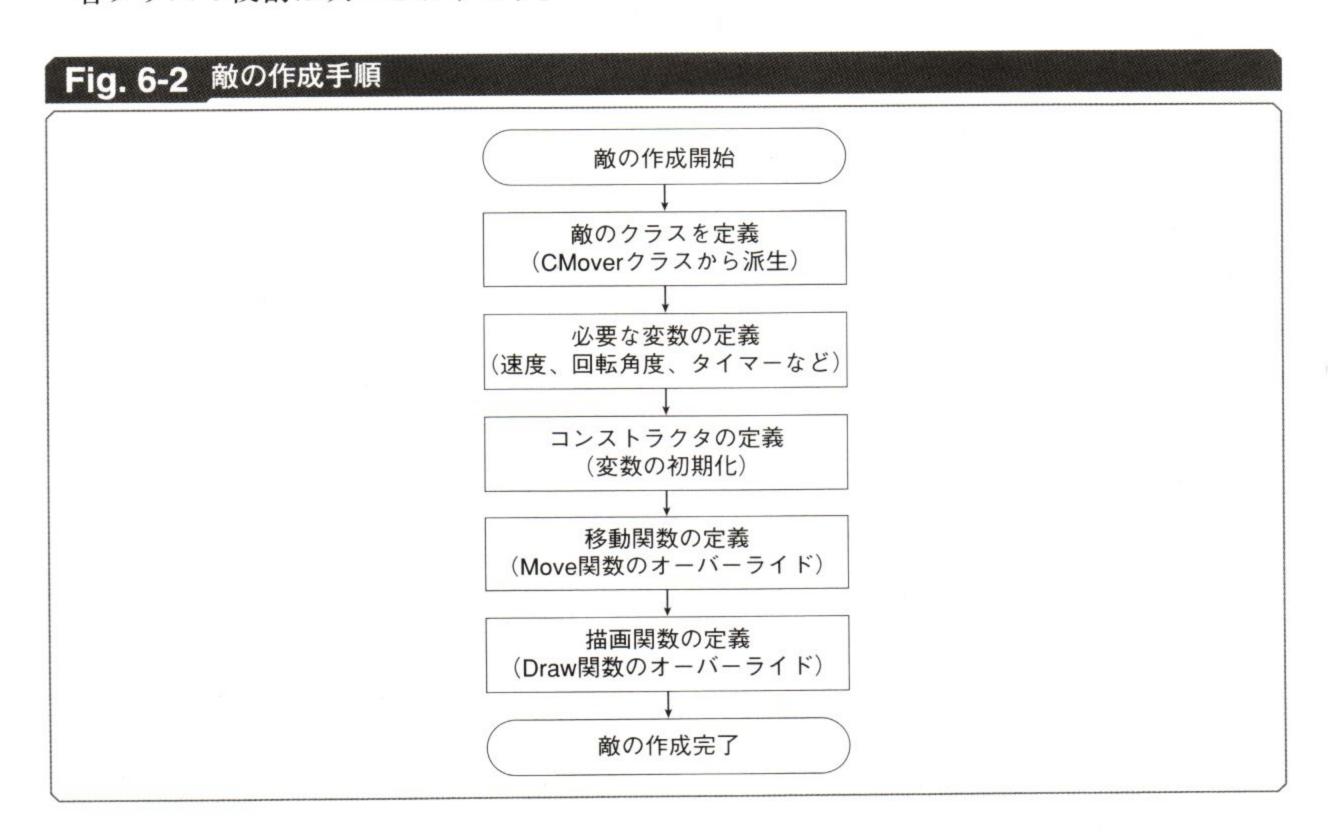
Fig. 6-2は敵の作成手順をまとめたものです。まず、移動物体クラス（CMover）の派生クラスとして、敵のクラスを定義します。次に、速度や回転角度、そしてタイマーといった必要な変数を定義します。最後に、移動処理（Move関数）と描画処理（Draw関数）をオーバーライドして、敵によって異なる移動処理と描画処理を記述すれば、新しい敵の完成です。

すべての敵には、耐久力やスコアといった共通の属性があります。これらの属性は個々の敵クラスで繰り返し実装するよりも、1つのクラスにまとめておき、それぞれの敵クラスをそこから派生させた方が効率的です。

そこで、敵の共通機能をまとめたクラス（CEnemy）を定義します。さらに、のちほどボスなどの大きさが異なる敵を作るために、小さなサイズの敵の共通機能をまとめたクラス（CSmallEnemy）を作ります（Fig. 6-3）。

赤身（CAkami）や玉子（CTamago）といった個々の敵クラスは、小さな敵のクラス（CSmallEnemy）から派生させます。これはちょうど、Chapter 5で弾クラス（CBullet）から方向弾クラス（CDirBullet）や狙い撃ち弾クラス（CAimBullet）を派生させたのと同じです（P. 170）。

各クラスの役割は次のとおりです。

**Fig. 6-2 敵の作成手順**

### CEnemyクラス

敵の共通機能をまとめたクラスです。CMoverクラスから派生します。

### CSmallEnemyクラス

小さな敵の共通機能をまとめたクラスです。CEnemyクラスから派生します。

### CAkamiクラス

寿司ネタの赤身をモチーフにした敵のクラスです。CSmallEnemyクラスから派生します。

### CTamagoクラス

寿司ネタの玉子をモチーフにした敵のクラスです。CSmallEnemyクラスから派生します。

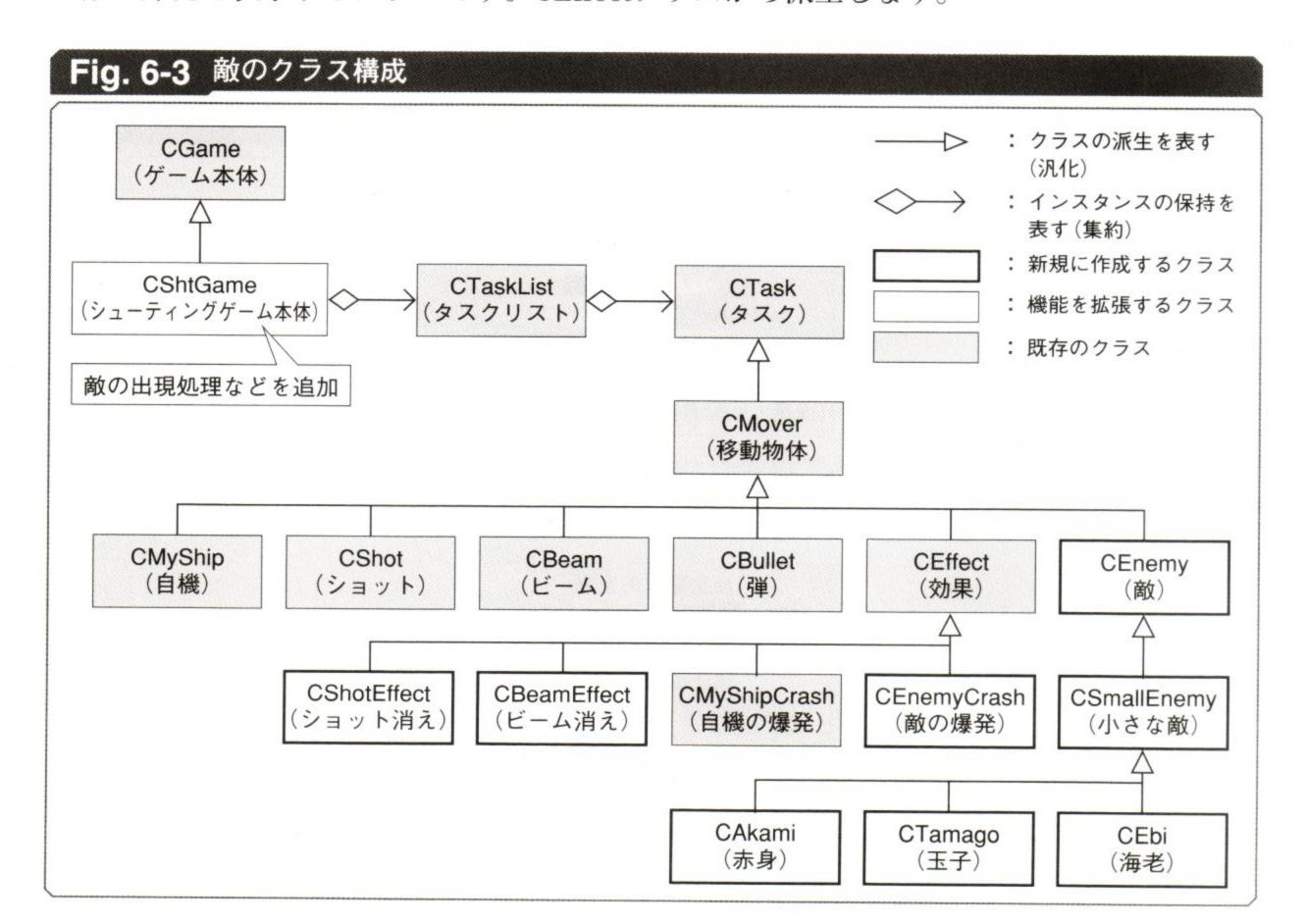
### CEbiクラス

寿司ネタの海老をモチーフにした敵のクラスです。CSmallEnemyクラスから派生します。

### CEnemyCrashクラス

敵の爆発を表示するクラスです。CEffectクラスから派生します。

**Fig. 6-3** 敵のクラス構成

### CShotEffectクラス

ショットが消えるエフェクトです。CEffectクラスから派生します。

### CBeamEffectクラス

ビームが消えるエフェクトです。CEffectクラスから派生します。

### CShtGameクラス

ゲーム本体のクラスです。敵の出現処理などを追加します。

## 敵の基本機能をまとめる

敵は、3Dモデル・回転角度・タイマー・耐久力・スコアといった属性を保持します。これらはすべての敵に共通する属性です。

List 6-1は、敵の基本機能をまとめたクラス（CEnemy）です。このクラスでは、Chapter 4の自機クラス（P. 122）やChapter 5の弾クラス（P. 150）と同様に、new演算子とdelete演算子をオーバーロードします。タスクリストには敵タスクリスト（EnemyList）を指定します。

**List 6-1　敵のクラス（Enemy.h、Enemy.cpp）**

```
// 敵の共通機能をまとめたクラス
// 移動物体クラス（CMover）から派生する
class CEnemy : public CMover {
protected:

    // 3Dモデル、回転角度、タイマー
    CMesh* Mesh;
    float Yaw;
    int Time;

public:

    // 耐久力、スコア
    float Vit;
    int Score;

    // 敵が発射する弾の色
    int Color;

    // new演算子、delete演算子
    void* operator new(size_t t) {
        return operator_new(t, Game->EnemyList);
    }
```

```
    void operator delete(void* p) {
        operator_delete(p, Game->EnemyList);
    }

    // コンストラクタ
    // 移動物体クラス(CMover)のコンストラクタを呼び出す
    CEnemy::CEnemy(
        CMesh* mesh,
        float x, float y, float l, float t, float r, float b,
        float vit, int score)
    :   CMover(Game->EnemyList, x, y, ENEMY_Z, l, t, r, b),
        Mesh(mesh), Yaw(0), Time(0), Vit(vit), Score(score)
    {
        Color=(int)(Game->Rand1()+0.5f);
    }

    // 3Dモデルの描画
    void DrawMesh(
        float x, float y, float z,
        float sx, float sy, float sz,
        float tx, float ty, float tz, TURN_ORDER to,
        float a, bool aa);
};

// 描画
// Chapter 2のメッシュクラス(P. 30)を利用
void CEnemy::DrawMesh(
    float x, float y, float z,
    float sx, float sy, float sz,
    float tx, float ty, float tz, TURN_ORDER to,
    float a, bool aa
) {
    Mesh->Draw(x, y, z, sx, sy, sz, tx, ty, tz, to, a, aa);
}
```

## 小さな敵の基本機能をまとめる

赤身や玉子といった、すべての小さな敵に共通する機能をまとめましょう。まず、小さな敵を生成するときには、座標・当たり判定・耐久力・スコアなどを初期化する必要があります。

当たり判定の設定方法は弾の場合と同様です(P. 148)。4つの数値を使って、敵の当たり判定の左端・上端・右端・下端の座標を表します。これらは敵の中心点からの相対座標です。それぞれの値を0に近くすると当たり判定が小さくなり、0から遠くすると当たり判

定が大きくなります。この当たり判定は、自機との接触判定の他、ショットやビームとの接触判定にも使用します。

小さな敵を移動する際には、敵が画面下方に消えたら敵を消去します。耐久力が0以下になったときには、スコアを加算して、爆発を表示します。スコアの加算については後述します(P. 205)。

3Dモデルを使って小さな敵を描画する場合には、3Dモデルを少し傾けて表示するとよいでしょう。サンプルでは画面手前方向に45度ほど傾けて描画しています。傾けない場合には3Dモデルを真上から見下ろした表示になるのですが、少し傾けると立体感が出ます。

List 6-2は、小さな敵のクラス(CSmallEnemy)です。

**List 6-2 小さな敵のクラス(Enemy.h、Enemy.cpp)**

```
// 小さな敵のクラス
// 敵クラス(CEnemy)から派生する
class CSmallEnemy : public CEnemy {
public:
    CSmallEnemy(CMesh* mesh, float x);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
// CEnemyコンストラクタの引数は、最初は3Dモデル、次の2つが座標、
// その次の4つが当たり判定、最後の2つが耐久力とスコア
CSmallEnemy::CSmallEnemy(CMesh* mesh, float x)
:   CEnemy(mesh, x, -54, -2.5f, -2.5f, 2.5f, 2.5f, 10, 100)
{}

// 移動
// 移動物体クラスの移動処理(CMover::Move関数)をオーバーライド
bool CSmallEnemy::Move() {

    // 耐久力が0以下になったらスコアを加算して爆発を表示
    if (Vit<=0) {
        Game->AddScore(Score);
        new CEnemyCrash(X, Y);
        return false;
    }

    // 画面下方に消えたらタスクを消去
    // 移動処理の戻り値をfalseにすると
    // 移動処理を呼び出す処理(MoveTasks関数)の働きによって
    // この敵のタスクは消去される
    return Y<50+8;
```

```
}

// 描画
// 移動物体クラスの描画処理(CMover::Draw関数)をオーバーライド
// List 6-1のDrawMesh関数を利用(P. 189)
void CSmallEnemy::Draw() {
    DrawMesh(X, Z, -Y, 1, 1, 1, 0.125f, Yaw, 0, TO_ZYX, 1, false);
}
```

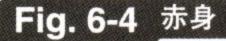
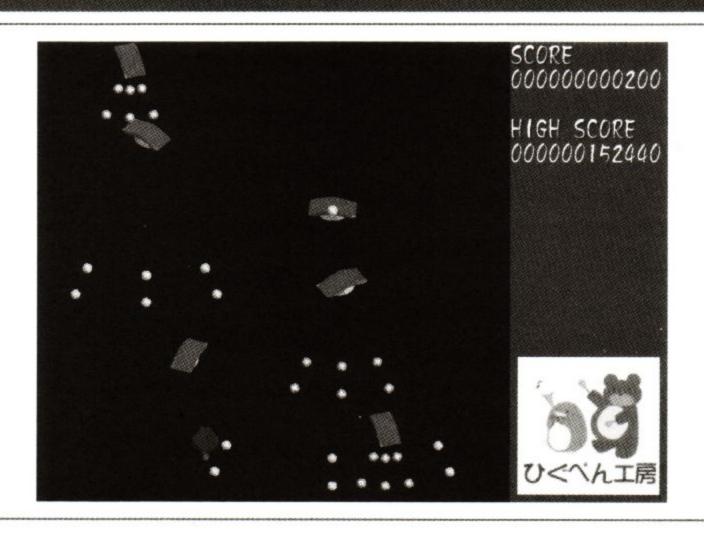
## いろいろな敵を作るには

ここでは赤身・玉子・海老といった敵を例に、小さな敵の作り方を解説します。より大きな敵を作る方法についてはChapter 8(P. 237)で、ボスを作る方法についてはChapter 9(P. 277)で解説します。

### 敵を作る：赤身の場合

最初は敵の1つである赤身を例に、敵の作り方を解説します。赤身は、画面上方から下方に向かって回転しながら直進し、ときどき3-way弾(3方向に広がる方向弾)を発射する敵です(Fig. 6-4)。

移動は、座標と回転角度を更新することによって実現しています。また、一定時間ごとに方向弾(P. 152)のタスクを生成します。発射方向を少しずつずらして3つの方向弾を発射すると、3-way弾になります。

Fig. 6-4 赤身

赤身の移動処理では、小さな敵に共通の移動処理（CSmallEnemy::Move関数）も実行します。共通の移動処理では、耐久力が0以下になったときの処理と、画面外に出たときの処理を行います（P. 150）。

List 6-3は、赤身に関する処理をまとめたクラス（CAkami）です。このクラスは、敵の共通処理をまとめた敵クラス（CEnemy）や小さな敵クラス（CSmallEnemy）から派生しています。これらのクラスから継承した変数や関数が利用できるので、赤身クラスに記述しなければならない処理は多くありません。

**List 6-3　赤身のクラス（Enemy.h、Enemy.cpp）**

```
// 赤身のクラス
class CAkami : public CSmallEnemy {
public:
    CAkami(float x);
    virtual bool Move();
};

// コンストラクタ
CAkami::CAkami(float x)
:   CSmallEnemy(Game->MeshAkami, x)
{}

// 移動
// 小さな敵クラスの移動処理をオーバーライドして
// 赤身独特の動きを記述する
// この移動処理はフレーム（約1/60秒）ごとに繰り返し呼び出される
bool CAkami::Move() {

    // 移動と回転
    Y+=0.4f;
    Yaw+=0.01f;

    // 一定時間ごとに弾を撃つ
    CMyShip* myship=Game->GetMyShip();
    if (myship && Time%10==0 && Time%100<20) {
        for (int i=-1; i<=1; i++) {
            new CDirBullet(
                Game->MeshBullet, Color,
                X, Y, 0.25f+i*0.04f, 1.0f, 0);
        }
    }
    Time++;

    // 小さな敵の共通処理を呼び出す
```

```
    return CSmallEnemy::Move();
}
```

## 敵を作る：玉子の場合

玉子は左右に曲がりながら飛来し、一定時間ごとに狙い撃ち弾を発射します（Fig. 6-5）。移動の処理は赤身クラス（CAkami）とは異なりますが、プログラムの基本的な構成は同じです。

List 6-4は、玉子の処理をまとめたクラス（CTamago）です。

**Fig. 6-5 玉子**

**List 6-4 玉子のクラス（Enemy.h、Enemy.cpp）**

```
// 玉子のクラス
class CTamago : public CSmallEnemy {
public:
    CTamago(float x);
    virtual bool Move();
};

// コンストラクタ
CTamago::CTamago(float x)
:   CSmallEnemy(Game->MeshTamago, x)
{}

// 移動
bool CTamago::Move() {
```

```
    // 移動と回転
    X+=sinf(Time*0.05f)*0.8f;
    Y+=0.3f;
    Yaw=-sinf(Time*0.03f)*0.1f;

    // 弾の発射
    CMyShip* myship=Game->GetMyShip();
    if (myship && Time%6==0 && Time%100<=20) {
        new CAimBullet(
            Game->MeshBullet, Color, X, Y,
            myship->X, myship->Y, 0, 1.0f, 0);
    }
    Time++;

    // 小さな敵の共通処理
    return CSmallEnemy::Move();
}
```

## 敵を作る：海老の場合

海老の場合には、生成時に乱数を使って進行方向をランダムに決め、速度を計算します。そして一定方向に直進し、ときどき分裂弾を発射します（Fig. 6-6）。プログラムの構成は赤身や玉子と同様です。

List 6-5は、海老の処理をまとめたクラス（CEbi）です。

Fig. 6-6 海老

**List 6-5　海老のクラス（Enemy.h、Enemy.cpp）**

```
// 海老のクラス
class CEbi : public CSmallEnemy {
    float VX, VY;
public:
    CEbi(float x);
    virtual bool Move();
};

// コンストラクタ
CEbi::CEbi(float x)
:   CSmallEnemy(Game->MeshEbi, x)
{
    // 進行方向をランダムに決める
    Yaw=Game->Rand05()*0.2f+0.25f;
    float rad=Yaw*D3DX_PI*2;
    VX=0.5f*cosf(rad);
    VY=0.5f*sinf(rad);
}

// 移動
bool CEbi::Move() {

    // 移動
    X+=VX;
    Y+=VY;

    // 弾の発射
    if (Game->Rand1()<0.04f) {
        new CSplitBullet(
            Game->MeshBullet, Color, X, Y, Yaw+0.5f, 0.4f);
    }

    // 小さな敵の共通処理
    return CSmallEnemy::Move();
}
```

## 敵の爆発

耐久力が0になった敵は爆発します。爆発の表示は、敵の爆発クラス（CEnemyCrash）のインスタンスを生成して実現します。

爆発の生成時には、爆発の効果音を再生します。そして、一定時間が経過したら爆発を消去します。

爆発の描画処理では、時間とともに拡大率とアルファ値を変化させながら、爆発の3Dモデルを描画します（P. 164）。これで、爆発がしだいに広がっていき、一定時間が経過すると消える効果が得られます。

List 6-6は、敵の爆発クラスです。

**List 6-6　敵の爆発（Effect.h、Effect.cpp）**

```
// 敵の爆発のクラス
class CEnemyCrash : public CEffect {
protected:

    // タイマー、拡大率
    int Time;
    float Scale;

public:

    // コンストラクタ、移動、描画
    CEnemyCrash(float x, float y, float scale=0.2f);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CEnemyCrash::CEnemyCrash(float x, float y, float scale)
:   CEffect(x, y), Time(0), Scale(scale)
{
    Game->PlaySE(Game->SECrash);
}

// 移動
// 一定時間が過ぎたら爆発を消去
bool CEnemyCrash::Move()
{
    Time++;
    return Time<=25;
}

// 描画
// 拡大率とアルファ値を変化させながら3Dモデルを描画
void CEnemyCrash::Draw() {
    float
```

```
        scale=Scale+Scale*Time,
        alpha=1.0f-0.04f*Time;
    Game->MeshCrashRice->Draw(
        X, Z, -Y, scale, 1, scale,
        0, 0, 0, TO_ZYX, alpha, false);
}
```

## ショットやビームと敵の当たり判定処理

敵をショットやビームで破壊できるようにするには、ショットやビームと敵との間で当たり判定処理を行う必要があります。当たり判定処理の結果、命中したと判定されたら、敵の耐久力を減少させます。

**Fig. 6-7 ショットやビームと敵の当たり判定処理**

ショットおよびビームの移動処理で
当たり判定処理を行う方法

ショットやビーム
から敵を参照する

ショット１の移動処理
（すべての敵との当たり判定処理を行う）

ショット２の移動処理
（すべての敵との当たり判定処理を行う）

ビーム１の移動処理
（すべての敵との当たり判定処理を行う）

敵１の移動処理

敵２の移動処理

敵の移動処理で
当たり判定処理を行う方法

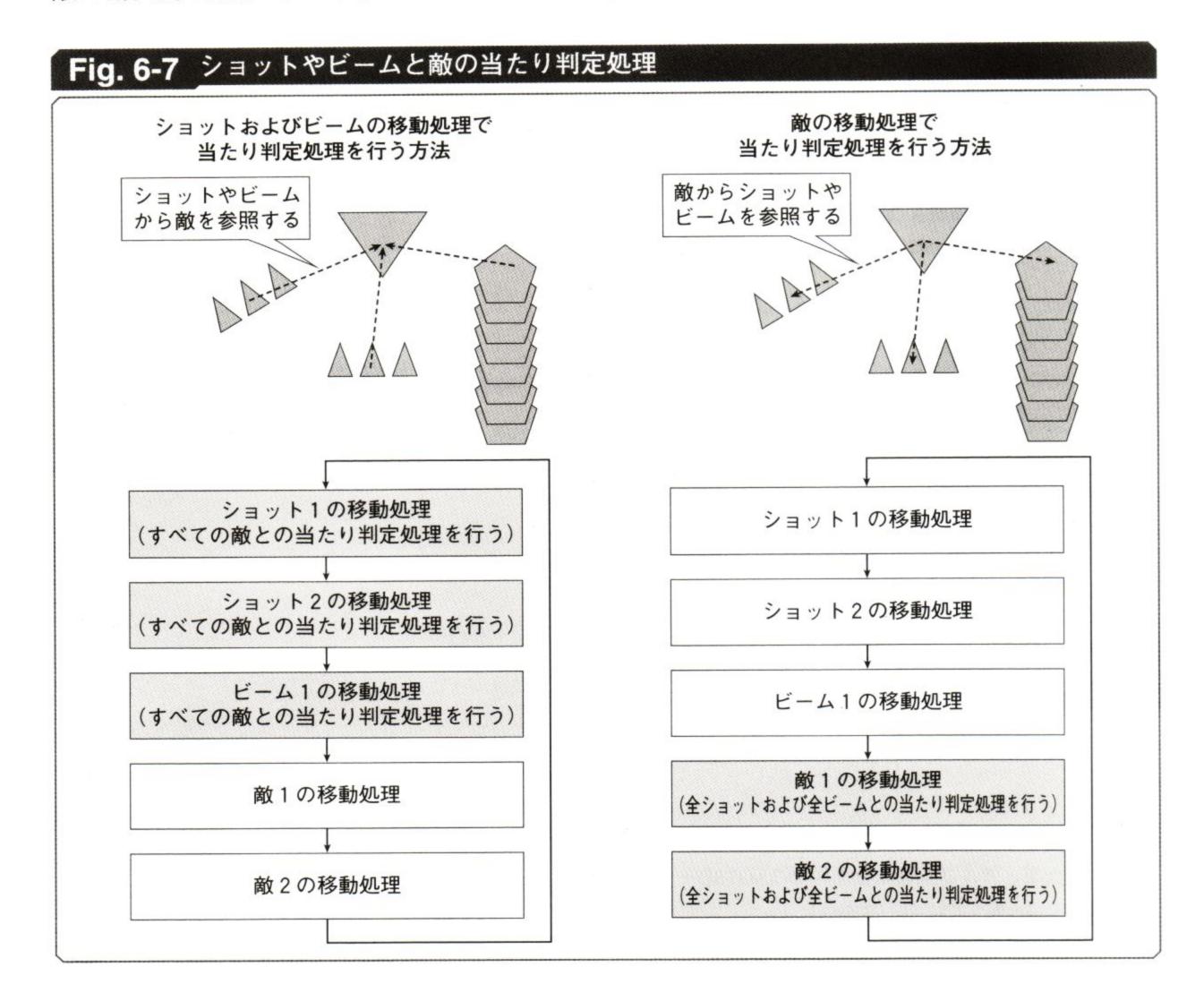
敵からショットや
ビームを参照する

ショット１の移動処理

ショット２の移動処理

ビーム１の移動処理

敵１の移動処理
（全ショットおよび全ビームとの当たり判定処理を行う）

敵２の移動処理
（全ショットおよび全ビームとの当たり判定処理を行う）

ショットやビームと敵との当たり判定処理は、ショットやビームの移動処理のなかで行う方法と、敵の移動処理のなかで行う方法とがあります。これは自機と弾との当たり判定処理を行う場合（P. 157）と同様です。それぞれの様子をFig. 6-7に示しました。

どちらの方法を使用しても結果はほぼ同じです。本書では、ショットやビームの移動処理のなかで判定処理を行うことにしました。

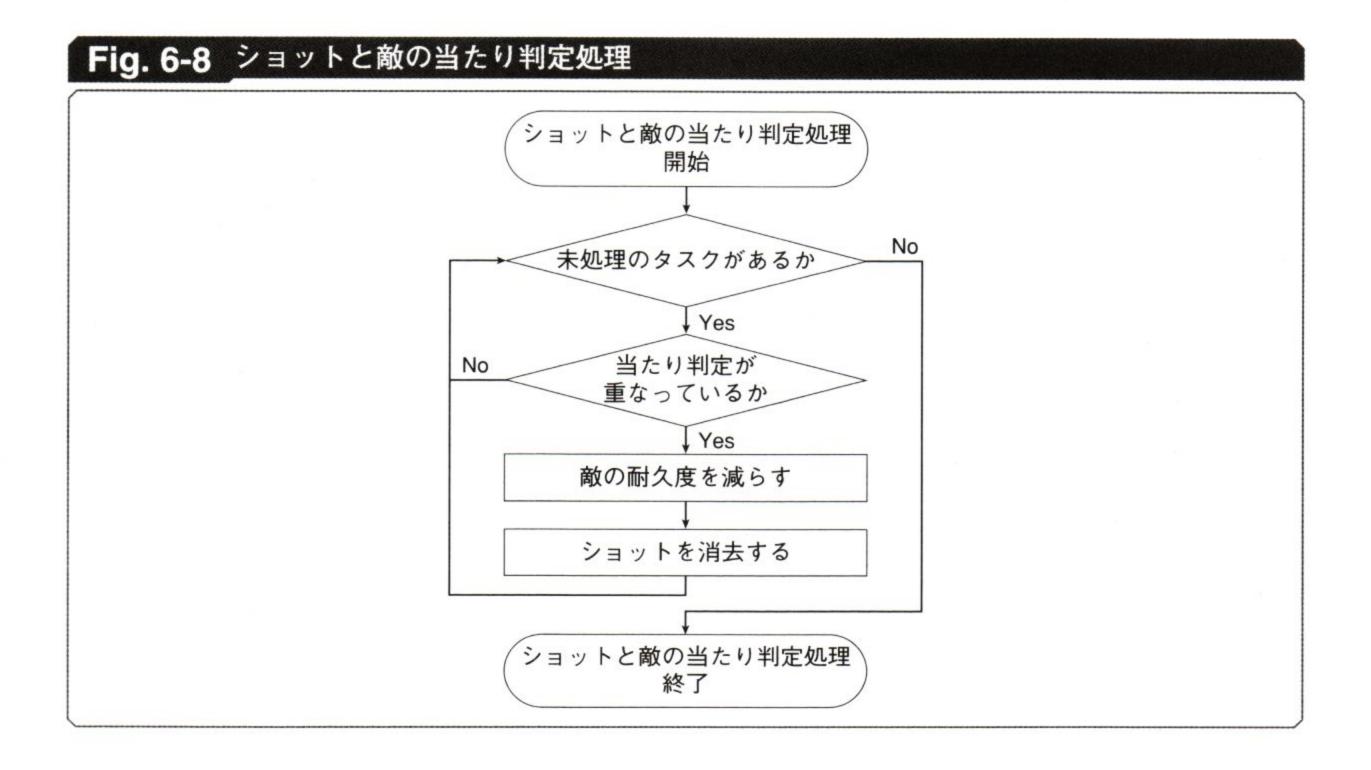
## ショットの当たり判定処理

ショットと敵の当たり判定処理を行うための手順をFig. 6-8にまとめました。

ショットと敵の当たり判定処理を行うには、Chapter 3で作成したタスクイテレータクラス（P. 97）を使って、すべての敵タスクについてループし、ショットと敵の接触を判定します。接触したときには、敵の耐久力を減らし、ショットを消去します。以上の処理をすべてのショットに対して行います。

また、接触時にはショットが消えるエフェクトも表示します。このエフェクトについては後述します（P. 202）。

List 6-7は、ショットの当たり判定処理です。Chapter 4で作成した移動処理（P. 116）に、敵との当たり判定処理を追加します。

**Fig. 6-8** ショットと敵の当たり判定処理

**List 6-7　ショットの当たり判定処理（MyShip.h、MyShip.cpp）**

```
bool CShot::Move() {

    // ショットの移動
    X+=VX;
    Y+=VY;

    // 当たり判定処理（ショットと敵）
    for (CTaskIter i(Game->EnemyList); i.HasNext(); ) {
        CEnemy* enemy=(CEnemy*)i.Next();

        // 敵に接触したときの処理
        // 接触判定には移動物体クラス（CMover）の
        // Hit関数を利用（P. 149）
        if (Hit(enemy)) {

            // 敵の耐久力を減らす
            enemy->Vit-=1;

            // ショットが消えるエフェクトを表示する
            for (int j=0; j<3; j++) new CShotEffect(X, Y, j);

            // ショットの消去
            return false;
        }
    }

    // ショットが画面外に出たら消去する
    return !Out(5);
}
```

## ビームの当たり判定処理

ビームの当たり判定処理もショットの場合と同様です。ビームと敵の当たり判定処理を行うには、タスクイテレータクラス（P. 97）を使って、すべてのタスクについてループし、ビームと敵の接触を判定します。接触したときには、敵の耐久力を減らし、ビームを消去します。また、ビームが消えるエフェクトも表示します。

List 6-8は、ビームの当たり判定処理です。ショットと同様に、Chapter 4で作成したビームの移動処理（P. 139）に対して、敵との当たり判定処理を追加します。

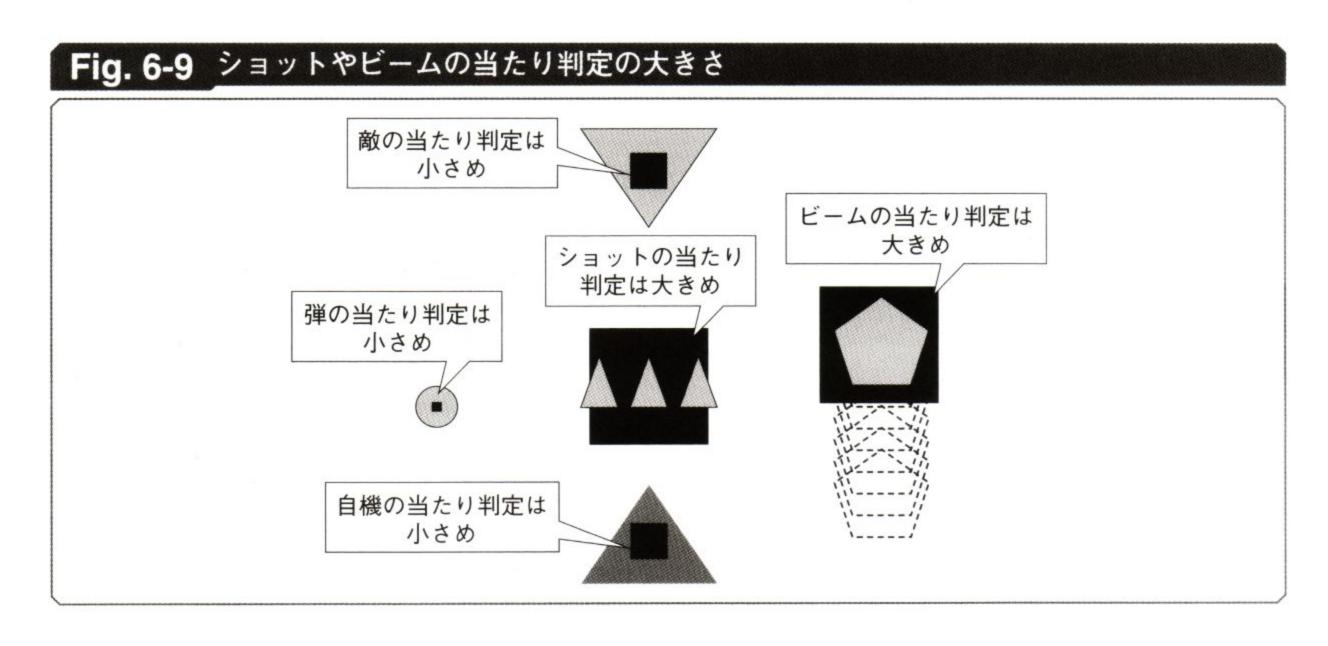
**List 6-8　ビームの当たり判定処理（MyShip.h、MyShip.cpp）**

```
// 移動
bool CBeam::Move() {

    // ビームの移動
    if (MyShip) {
        X=MyShip->X;
    } else {
        return false;
    }
    Y-=6.0f;

    // 当たり判定処理（ビームと敵）
    for (CTaskIter i(Game->EnemyList); i.HasNext(); ) {
        CEnemy* enemy=(CEnemy*)i.Next();

        // 敵に接触したときの処理
        // 移動物体クラス（CMover）のHit関数を利用（P. 149）
        if (Hit(enemy)) {

            // 敵の耐久力を減らす
            enemy->Vit-=4;

            // ビームが消えるエフェクトを表示する
            for (int i=0; i<4; i++) new CBeamEffect(X, Y);

            // ビームの消去
            return false;
        }
    }

    // ビームが画面外に出たら消去する
    return !Out(8);
}
```

## 当たり判定の大きさ

ショットやビームの当たり判定は大きめにする必要があります。敵の当たり判定を小さめに設定してあるので、ショットやビームの当たり判定を大きくしないと、敵に命中させることが難しいからです（Fig. 6-9）。

敵の当たり判定を小さめにしてあるのは、自機が敵を避けやすくするためです。個性的

**Fig. 6-9 ショットやビームの当たり判定の大きさ**

な敵キャラクターを作成したら、画面上に派手に登場させてプレイヤーに印象づけたいところです。しかし、敵が見た目どおりの当たり判定を持っている場合、大きめの敵を登場させると自機が動ける範囲がすぐに狭まってしまいます。ゲームとして成立しなくなるケースもあるでしょう。そこで敵の判定を小さくすることによって、自機が敵をかすめ飛ぶような動きを可能にしています。

自機に対する当たり判定と、ショットやビームに対する当たり判定とを別々に設定しておく方法もありますが、ここでは処理を簡単にするため、同じ当たり判定を両方の用途に使っています。そのかわりに、自機の当たり判定は小さめに、ショットやビームの当たり判定は大きめに設定したわけです。

# ショットとビームのエフェクト

ショットやビームは敵に命中すると消えますが、このときに消滅のエフェクトを表示すると、かなり見た目がよくなります。特に、ボスなどの大きな敵は耐久力が高いので、ショットやビームで撃墜するまでに少し時間がかかります。この場合にショットやビームが消滅するエフェクトがあると、敵の耐久力を削っている雰囲気を出すことができます。

本書のサンプルにも、ショットやビームが消滅する際に液体（醤油？）が飛び散るエフェクトを追加してみました（Fig. 6-10、11）。

このエフェクトは、拡大率とアルファ値を変化させながら液体の3Dモデルを表示することによって実現しています。

まず、エフェクトの生成時に乱数を使って、飛沫の飛ぶ速度をランダムに決めます。そして、移動時には座標・拡大率・アルファ値を更新します。また、一定時間が経過したらエフェクトを消去します。

描画処理では、計算した座標・拡大率・アルファ値を用いて、3Dモデルを描画します。結果として、飛沫が拡大率やアルファ値を変化させながら飛び散るエフェクトが得られます。

List 6-9は、ショット消滅時のエフェクトを表すクラス（CShotEffect）です。ビーム消滅時のエフェクト（CBeamEffect）もほとんど同様のプログラムで実現しています。詳細は付録CD-ROMの「Effect.h」および「Effect.cpp」をご覧ください。

**Fig. 6-10** ショットが消滅するエフェクト

**Fig. 6-11** ビームが消滅するエフェクト

**List 6-9　ショット消滅時のエフェクト（Effect.h、Effect.cpp）**

```
// ショット消滅時エフェクトのクラス
class CShotEffect : public CEffect {
protected:

    // 速度、拡大率、アルファ値、タイマー
    float VX, VY, Scale, Alpha;
    int Time;

public:

    // コンストラクタ、移動、描画
    CShotEffect(float x, float y, int delay);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CShotEffect::CShotEffect(float x, float y, int delay)
:   CEffect(x, y-4), Scale(0.6f), Alpha(1.0f), Time(-delay*5)
{
    float rad=(0.25f+0.4f*Game->Rand05())*1.8f*D3DX_PI;
    VX=2.0f*cos(rad);
    VY=2.0f*sin(rad);
}

// 移動
bool CShotEffect::Move() {
    X+=VX;
    Y+=VY;
    Scale-=0.015f;
    Alpha-=0.025f;
    Time++;
    return Time<40;
}

// 描画
void CShotEffect::Draw() {
    Game->MeshCrashSauce->Draw(
        X, Z, -Y, Scale, Scale, Scale,
        0, 0, 0, TO_NONE, Alpha, false);
}
```

# スコアの加算と表示

多くのシューティングゲームでは、敵を破壊するとスコア（得点）が加算されます。そこで、今度はスコアの加算や表示を行う処理について解説します。

スコアの保持には、数値を表すintなどの既存の型を使用することもできます。しかし、非常に桁数の多いスコアに対応したり、スコアを効率よく表示したりするには、桁数の多い数を扱うためのクラスを用意した方が便利です。ここでは桁数の多い数を扱うためのクラス（CBigNum）を作成しました。このクラスについては後述します（P. 206）。

スコアに関する処理はゲーム本体に追加します。まず必要なのは、スコアとハイスコアを保持する変数です。スコアは、ゲームの開始時には0にしておきます。ハイスコアはファイルに保存しておき、ゲームの起動時に読み込むとよいでしょう。

次に、スコアの加算処理を用意します。ここではスコアを加算するとともに、スコアとハイスコアを比較して、スコアがハイスコアを上回った場合には現在のスコアを新たなハイスコアとします。

List 6-10は、スコアに関する処理を追加したゲーム本体クラスです。

**List 6-10　スコアの処理（Main.h）**

```
class CShtGame : public CGame {

    // ...（中略）...

    // スコアとハイスコアを保持する変数
    CBigNum *Score, *HighScore;

    // スコアの加算
    void AddScore(int x) {

        // 引数で指定された値をスコアに加算する
        Score->Add(x);

        // スコアがハイスコアを上回った場合にはハイスコアを更新する
        if (Score->Compare(HighScore)>0) HighScore->Set(Score);
    }
}
```

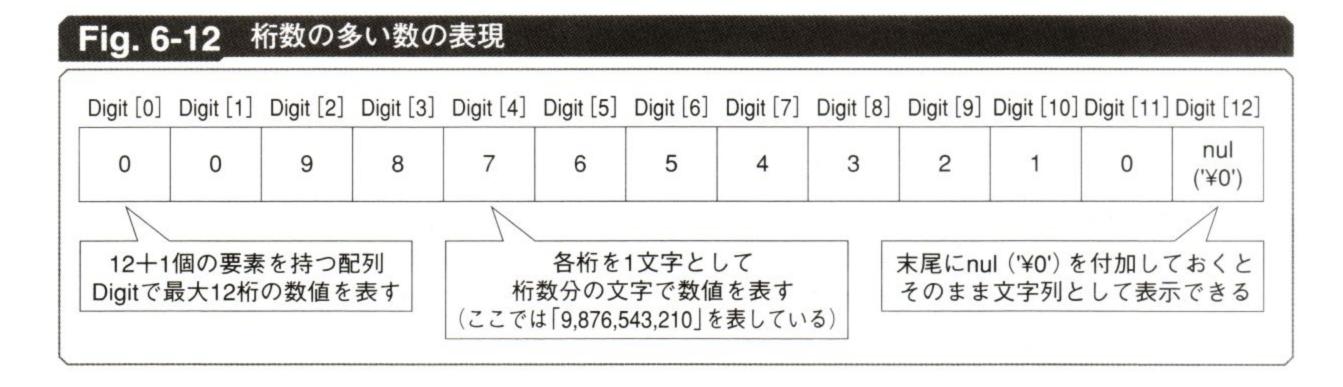
## 桁数の多い数を扱うには

ハードウェアで直接に演算が行える数の範囲は決まっています。その制限よりも多くの桁数を持つ数を演算に使用するには、多倍長演算という手法を使います。

サンプルでは、数値の1桁を1文字で表すことによって、桁数の多い数を表現しました（Fig. 6-12）。文字を保持する配列（Fig. 6-12では配列Digit）を長くすれば、どんなに桁数の多い数でも表せるという仕組みです。

科学技術計算などに使う多倍長演算では、1桁を1文字で表すのではなく、より多くの桁を1つの数値で表すことによって処理を高速化します。しかし、今回のようなゲームの場合には、ここで解説するように1桁を1文字で表した方が、効率よくスコアを表示することができます。

なお、Fig. 6-12のように配列の最後にnul（'¥0'）を付加しておくと、スコアの表示が簡単になります。配列をそのまま文字列として扱うことができるからです。

**Fig. 6-12** 桁数の多い数の表現

## 大きな数を扱うための仕組み

本書では、多倍長演算の機能をまとめたクラス（CBigNum）を作成しました。このクラスは桁数と各桁の数値をメンバ変数で保持します。桁数はコンストラクタで指定します。

このクラスには次のような関数を用意します。

### 数値を設定する関数（Set）

スコアの初期化などに使用します。引数の型に応じて、何種類かの関数を用意しています。

### 比較を行う関数（Compare）

スコアとハイスコアを比較するときなどに使用します。

### 加算を行う関数（Add）

スコアの加算に使用します。また、乗算を行う関数（Mul）も用意します。

### 数値と桁数を取得する関数 (GetDigits、GetNumDigits)

スコアの表示に使用します。数値を取得する関数 (GetDigits) は、数値を保持する配列 (Digits) を返します。前述のように、この配列の末尾にはnulが付加されているため、文字列として扱うことができます。

※

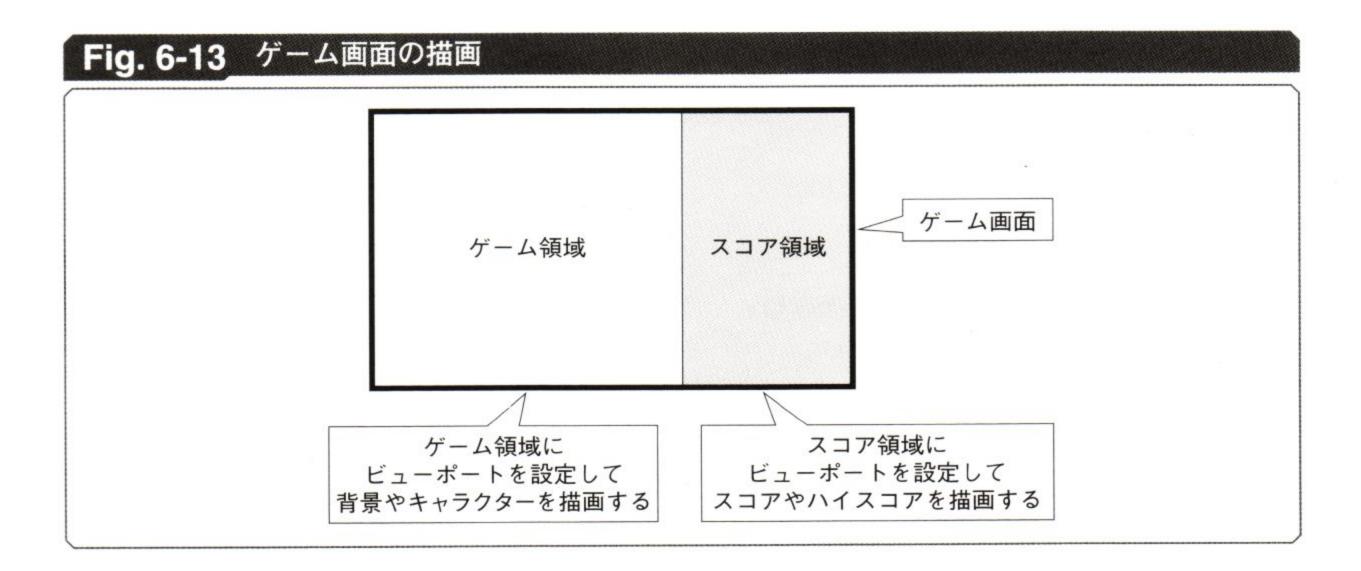
各演算の具体的な処理に関しては付録CD-ROMの「BigNum.h」と「BigNum.cpp」をご参照ください。

## スコアの表示

本書のサンプルでは、ゲーム画面を「ゲーム領域」と「スコア領域」の2つに分けています。ゲーム領域にはゲームの背景やキャラクターを主に3Dグラフィックで描画し、スコア領域にはスコアやハイスコアなどの情報を主に2Dグラフィックで描画します。一般のシューティングゲームでも、このように画面を分割して使用している例が多く見られます。

ゲーム領域とスコア領域では描画方法が異なるので、ビューポートという機能を使って、これらの領域を別々に描画します (Fig. 6-13)。ビューポートを使うと、決められた領域だけに描画を行い、はみ出した部分はカットさせることができます。

画面を描画するときには、まず画面の左側にゲーム領域となるビューポートを設定します。そして、自機・弾・敵などのキャラクターを描画します。

**Fig. 6-13 ゲーム画面の描画**

次に、画面の右側にスコア領域となるビューポートを設定します。そして、スコアとハイスコアを表示します。文字の表示にはChapter 2のフォントクラス（P. 35）を使います。

本書では制作元（筆者）のロゴ画像も表示することにしました。画像の表示にはChapter 2のテクスチャクラス（P. 32）を使います。同じ方法でロゴ以外の画像を表示することもできます。ゲームのタイトルを表示したり、キャラクターの絵を表示したりなど、いろいろと工夫してみてください。

なお、スコア領域にはフレーム落ちに関する情報も表示しています。本章のサンプルでは、Chapter 2のゲーム本体クラス（P. 20）の機能によって、処理が重くなったときに処理落ちするかフレーム落ちするかを選ぶことができます。ゲーム中にキーボードの「B」キーかジョイスティックのボタン「4」を押すと、処理落ちとフレーム落ちが切り替わります。フレーム落ちを選択しているときには、スコア領域に「DROP FRAMES」と表示します。

List 6-11は、スコアの表示に関するプログラムです。スコアの表示処理は、ゲーム画面全体の描画処理（CShtGame::Draw関数）のなかに記述します。

**List 6-11　スコアの表示（Main.cpp）**

```
void CShtGame::Draw() {
    LPDIRECT3DDEVICE9 device=Graphics->GetDevice();

    // ゲーム領域の初期化
    D3DVIEWPORT9 viewport;
    int w, h;
    w=Graphics->GetWidth();
    h=Graphics->GetHeight();
    viewport.X=0;
    viewport.Y=0;
    viewport.Width=h;
    viewport.Height=h;
    viewport.MinZ=0;
    viewport.MaxZ=1;
    device->SetViewport(&viewport);
    Graphics->Clear(ColBlack);

    // キャラクターの描画
    DrawTask(EffectList);
    DrawTask(ShotList);
    DrawTask(BeamList);
    DrawTask(MyShipList);
    DrawTask(EnemyList);
    DrawTask(BulletList);
```

```
        // スコア領域の初期化
        viewport.X=h;
        viewport.Width=w-h;
        device->SetViewport(&viewport);
        Graphics->Clear(D3DCOLOR_XRGB(100, 50, 80));

        // スコア、ハイスコアの描画
        Font->DrawText("SCORE", h, 0, ColWhite, ColShade);
        Font->DrawText(Score->GetDigits(), h, 32, ColWhite);
        Font->DrawText("HIGH SCORE", h, 96, ColWhite, ColShade);
        Font->DrawText(HighScore->GetDigits(), h, 128, ColWhite);

        // ロゴの描画
        int margin=(w-h-ImageLogo->GetOriginalWidth())/2;
        ImageLogo->Draw(
            h+margin, h-ImageLogo->GetOriginalHeight()-margin,
            ColWhite);

        // フレーム落ちを行うかどうかの表示
        if (Game->DropFrames) {
            Font->DrawText("DROP FRAMES", h, 560, ColWhite, ColShade);
        }
}
```

## ハイスコアの保存

多くのゲームは、ハイスコアをファイルに保存する機能を備えています。そこで、本書のサンプルにもハイスコア保存機能を用意してみました。実行ファイル1つ上のフォルダにある「ShtGame.ini」というファイルに、ハイスコアを次のような文字列で保存します。

**HighScore=000000014900**

一般には、ハイスコアをテキストエディタなどで勝手に書き換えられないように、なんらかの暗号化を施します。しかし、ここではわかりやすくするために、そのままテキストで保存することにしました。

ハイスコアをファイルに記録する処理は、プログラムの終了時に行います。逆にプログラムの開始時には、ファイルに記録したハイスコアを取得します。ハイスコアがまだ記録されていない場合には、ハイスコアを0に設定します。これらの処理は、例えばゲーム本体のデストラクタとコンストラクタで行うとよいでしょう。

List 6-12は、ハイスコアの保存に関するプログラムです。ファイルの読み書きには、

「LibUtil¥Util.h」と「LibUtil¥Util.cpp」で定義した文字列操作クラス（CStrings）を使いました。ここはもちろん他の適当なAPIを使ってもかまいません。

**List 6-12　ハイスコアの保存（Main.cpp）**

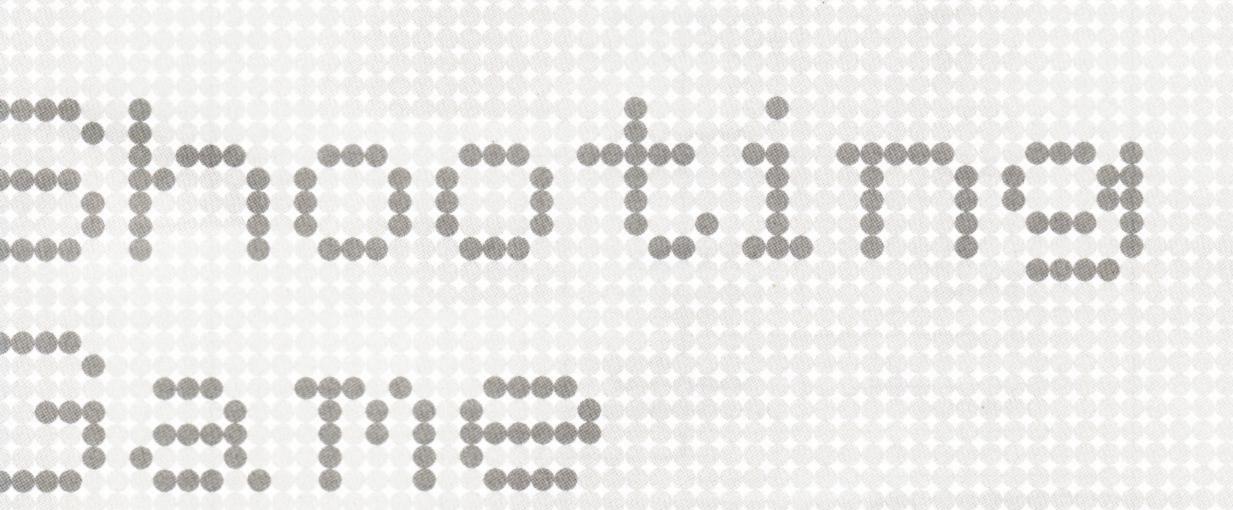
```
// デストラクタ
CshtGame::~CshtGame() {

    // ハイスコアの記録
    CStrings* ss=new CStrings();
    ss->SetValue("HighScore", HighScore->GetDigits());
    ss->SaveToFile(ExePath+"..¥¥ShtGame.ini");
    delete ss;
}

// コンストラクタ
CMurasame::CMurasame()

// ...（中略）...
{
    // ...（中略）...

    // スコアの初期化
    Score=new CBigNum(12);
    HighScore=new CBigNum(12);

    // ハイスコアの取得
    CStrings* ss=new CStrings();
    ss->LoadFromFile(ExePath+"..¥¥ShtGame.ini");
    string s=ss->GetValue(string("HighScore"));
    if (!s.empty()) HighScore->Set(s);
    delete ss;
}
```

## >>Chapter 6のまとめ

本章では敵の移動と表示、敵が弾を撃つ処理、ショットやビームと敵との当たり判定処理、得点の加算と表示について解説しました。これで、ゲームの核となる要素はおおむねそろってきました。そこで次章では、ゲームの外枠となる、タイトル画面、ゲームオーバー画面、残機の管理、コンティニュー画面などについて説明します。

# Chapter 07

# ゲームの外枠

シューティングゲームを構成する要素のうち、遊ぶための中核的な部分は前章までに作成しました。そこで本章では、ゲームの体裁を整え、遊びやすく、そして魅力的に仕上げるための要素について考えてみましょう。
具体的には、タイトル画面、レディ画面、ポーズ画面、残機の管理、コンティニュー画面、ゲームオーバー画面といったゲームの外枠となる部分の作り方を解説します。

# 画面間の遷移

ほとんどのゲームには、ゲーム本編の画面以外にも、タイトル画面やゲームオーバー画面といったさまざまな種類の画面があります。画面の構成はゲームによって違いますが、こういった複数の画面間を遷移しながら進行するという構造は、多くのゲームに共通です。

本書のサンプルのように、タスクシステムをベースにしたプログラムでは、タスクの生成と消去を使って画面間の遷移を実現します。タスクの生成は新しい画面の生成に、タスクの消去は古い画面の消去に対応します。古い画面から新しい画面へ遷移するには、前者を消去して、後者を生成します。

Fig. 7-1は画面関連のタスクの生成関係を表した図です。図中の「タイトル画面」や「レディ画面」などは画面の種類を、「CTitle」や「CReady」などはその画面に対応するクラスを示します。例えば、タイトル画面でゲームをスタートすると、レディ画面（「GET READY」と表示する画面）のタスクと、出現時の自機のタスク（画面外から登場する自機）が生成されます。

本章のプロジェクトは付録CD-ROMの「ShtGame_Scene」フォルダに収録しました。実行ファイルは「ShtGame_Scene¥Release¥ShtGame.exe」です。

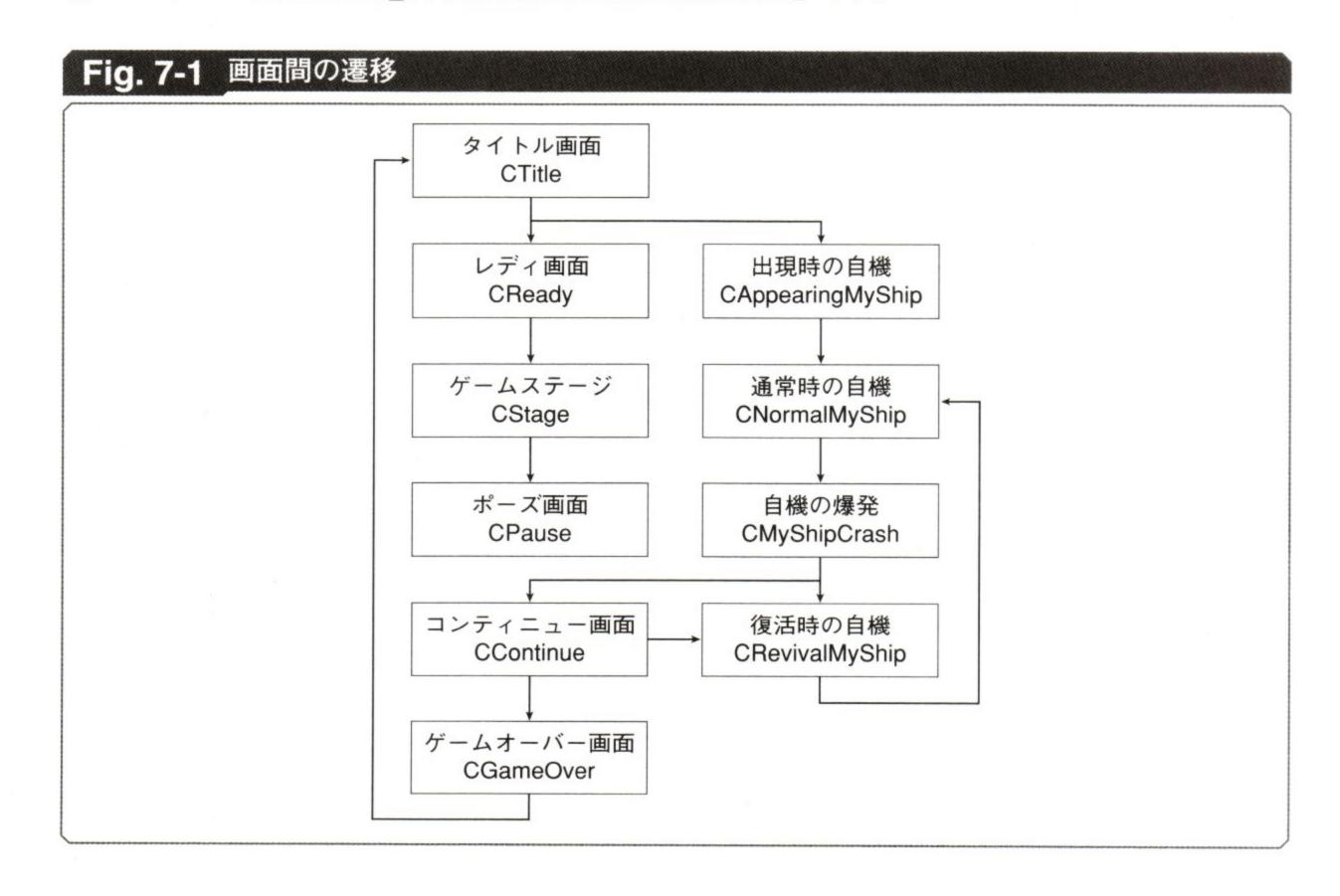
**Fig. 7-1** 画面間の遷移

# クラス構成

Fig. 7-2に本章で作成または変更するクラスの構成を示しました。各クラスの役割は以下のとおりです。

## CScene

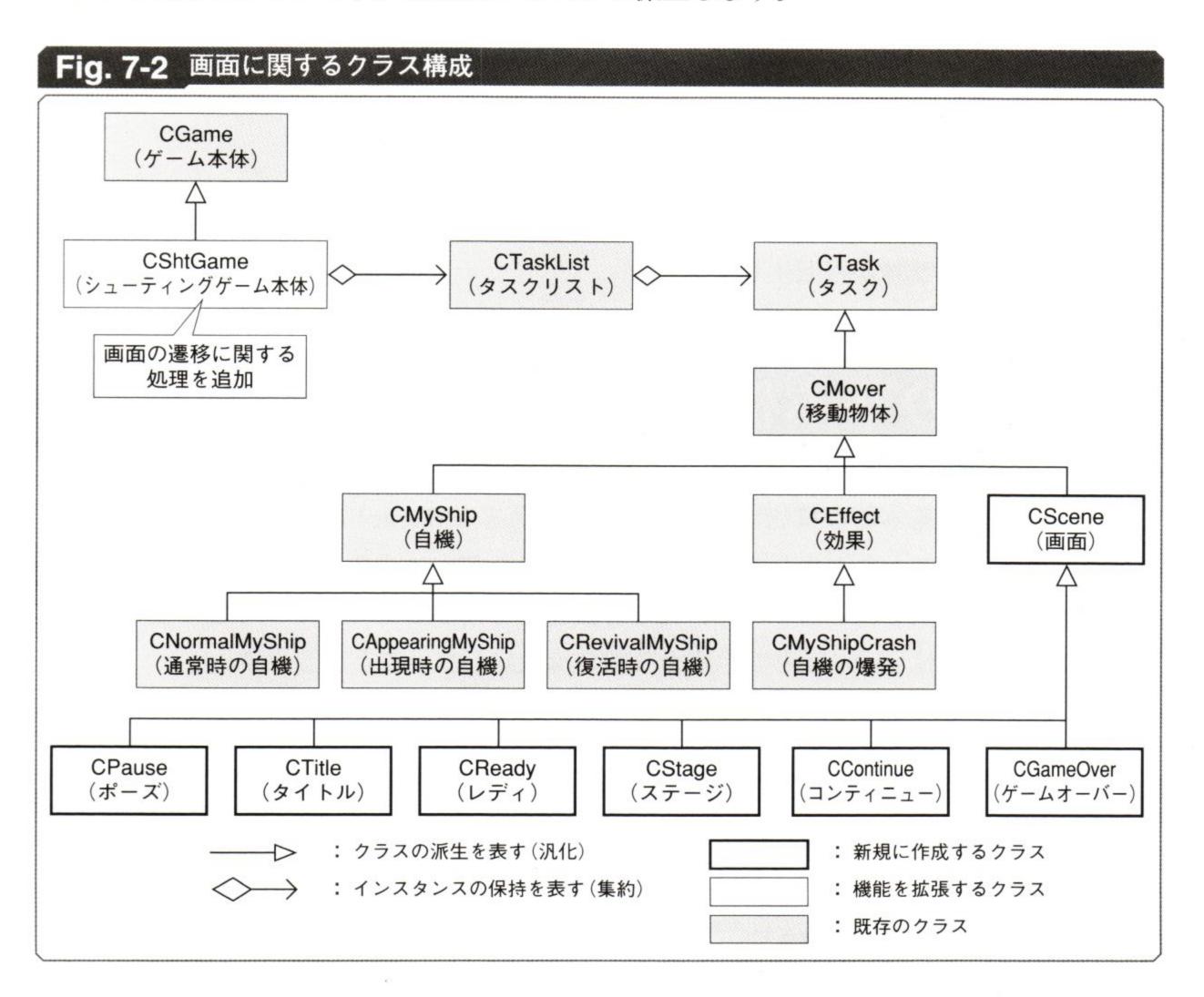
各種の画面に関する基本的な機能をまとめたクラスです。CMoverクラスから派生します。

## CTitle

タイトル画面のクラスです。CSceneクラスから派生します。

## CReady

レディ画面のクラスです。CSceneクラスから派生します。

Fig. 7-2 画面に関するクラス構成

### CStage
ゲームのステージを表すクラスです。CSceneクラスから派生します。

### CPause
ポーズ画面のクラスです。CSceneから派生します。

### CContinue
コンティニュー画面のクラスです。CSceneクラスから派生します。

### CGameOver
ゲームオーバー画面のクラスです。CSceneクラスから派生します。

### CAppearingMyShip
出現時の自機を表すクラスです（P. 169）。CMyShipクラスから派生します。

### CRevivalMyShip
復活時の自機を表すクラスです（P. 167）。CMyShipクラスから派生します。

### CShtGame
シューティングゲーム本体を表すクラスです。本章では画面の遷移に必要な機能を追加します。CGameクラスから派生します。

## 画面の基本機能

各画面に関する基本的な機能は、画面の基本クラス（CScene）にまとめます。このクラスは移動物体クラスから派生しますが、座標や当たり判定は必要ありません。自機や弾などとは違い、画面内を動き回ったり、他の移動物体と接触したりするわけではないからです。そこで、座標は（0, 0, 0）とし、当たり判定は設定しないことにします。

List 7-1は、画面の基本クラスです。

**List 7-1　画面の基本機能（Scene.h）**

```
class CScene : public CMover {
public:

    // new演算子、delete演算子
    // SceneListは画面用タスクリスト
```

```
    void* operator new(size_t t) {
        return operator_new(t, Game->SceneList);
    }
    void operator delete(void* p) {
        operator_delete(p, Game->SceneList);
    }

    // コンストラクタ
    CScene()
    :   CMover(Game->SceneList, 0, 0, 0)
    {}
};
```

## タイトル画面

多くのゲームにはタイトル画面があります。アーケードゲームの場合、タイトル画面やそれに続くデモ画面は、ゲームの魅力をアピールして、プレイヤーにコインを投入してもらうための重要な要素です。そのため、コインを投入する前のタイトルやデモなどの一連の画面をアドバタイズ画面（advertise：広告する、advertisement：広告）と呼ぶことがあります。

PCゲームの場合には、アドバタイズ画面を見てゲームの品定めをしてからコインを投入するわけではないので、それほど凝ったタイトル画面やデモ画面は必要ではありません。もちろん、アーケードゲームの雰囲気を再現するために、あえて凝ったアドバタイズ画面を作る場合もあります。また、プレイのコツをプレイヤーに伝える場として利用してもよいでしょう。

ここではシンプルに、ロゴとメニューだけのタイトル画面を作ります（Fig. 7-3）。このタイトル画面には「紫雨」というロゴと、モードを選択するためのメニューがあります。

メニュー項目の「START」はゲームの開始、「EXIT」はプログラムの終了です。項目はカーソルキーまたはジョイスティックの上下で選択し、「Z」キーまたはボタン「0」で決定します（「C」キーとボタン「2」にも対応しています）。

タイトル画面のポイントは、メニュー項目の選択や実行に関する処理です。キーボードやジョイスティックの入力に応じてメニュー項目を選択し、ボタンが入力されたら選択中の項目を実行します。

「START」を選択した場合には、後述するゲーム開始処理（P. 218）を呼び出します。ゲームを開始するための初期化処理が終わったら、タイトル画面のタスクを消去します。

描画処理では、「紫雨」のロゴとメニュー項目を表示します。メニューを描画する際には、選択中の項目は明るく、選択していない項目は暗く表示します。

List 7-2は、タイトル画面に相当するクラス（CTitle）です。

**Fig. 7-3 タイトル画面**

**List 7-2 タイトル画面（Scene.h、Scene.cpp）**

```
// タイトル画面のクラス
// 画面の基本クラス（CScene）から派生させる
class CTitle : public CScene {
protected:

    // メニュー項目の選択位置、前回のキー入力
    int MenuPos;
    bool PrevInput;

public:

    // コンストラクタ、移動、描画
    CTitle();
    virtual bool Move();
    virtual void Draw();
};

// メニュー項目のID
enum {
    MENU_START, MENU_EXIT, NUM_MENU_ITEMS
};

// コンストラクタ
```

```
// 最初はSTARTが選択されているように
// メニュー項目の選択位置を初期化する
CTitle::CTitle()
:   MenuPos(MENU_START)
{}

// 移動
// メニュー項目の選択や実行を行う
bool CTitle::Move() {

    // キーとジョイスティックの入力を調べる
    const CInputState* is=Game->GetInput()->GetState(0);

    // 前回の入力がなかった場合だけ操作を受け付ける
    if (!PrevInput) {

        // メニュー項目の選択
        if (is->Up && MenuPos>0) MenuPos--;
        if (is->Down && MenuPos<NUM_MENU_ITEMS-1) MenuPos++;

        // メニュー項目の実行
        if (is->Button[0]||is->Button[2]) {
            switch (MenuPos) {

                // ゲーム開始
                case MENU_START:
                    Game->Start();
                    return false;
                    break;

                // プログラム終了
                case MENU_EXIT:
                    PostQuitMessage(0);
                    break;
            }
        }
    }

    // 前回の入力を保存
    PrevInput=is->Up||is->Down||is->Button[0]||is->Button[2];
    return true;
}

// 描画
void CTitle::Draw() {
    int h=Game->GetGraphics()->GetHeight();
```

```
    D3DCOLOR w=ColWhite, g=D3DCOLOR_XRGB(64, 64, 64);

    // ロゴ
    Game->ImageTitle->Draw(h/2-298, h/2-298, g);
    Game->ImageTitle->Draw(h/2-300, h/2-300, w);

    // メニュー
    CFont* font=Game->Font;
    font->DrawText(
        "START", h/2-200, h/2+90, MenuPos==MENU_START?w:g);
    font->DrawText(
        "EXIT", h/2-200, h/2+122, MenuPos==MENU_EXIT?w:g);
}
```

## メニューの選択

このタイトル画面にあるようなメニューは、一般に上下キーで項目を選択して、決定キーで項目を実行します。ここで注意しなければならないのは、キーやボタンを押しっぱなしにした場合への対応です。

例えば、単に「下キーが入力されたときには次のメニュー項目に移動する」という処理だけでは、下キーを押したときに、選択位置がいちばん下のメニュー項目まで一気に移動してしまいます。キーを押しっぱなしにしたつもりがなくても、プログラムの処理が速いため、「キーが押された」ということが何度も検出されてしまうからです。

これを回避するには、前回の入力状態を保存しておき、キーやボタンが一度無入力状態になったことを確認した後に次の操作を受け付けます。List 7-2では、入力状態をメンバ変数(PrevInput)に保存しています。

なお、ここではメニュー項目がSTARTとEXITの2つだけなので、前回の入力を考慮しなくても問題は起きません。しかし、メニュー項目が増えたときに備えて、こういった処理を入れておくことにします。

## ゲームの開始

メニュー項目のSTARTが実行された場合には、ゲームの本編を開始します。

ゲーム開始時には、スコア・残機・コンティニュー回数・タイマーなどを初期化します。そして、出現時の自機と、後述するレディ画面のタスクを生成します。

ゲームを開始するための初期化処理は、List 7-3のようなプログラムで行います。

**List 7-3　ゲームの開始（Main.cpp）**

```
void CShtGame::Start() {

    // スコア、残機、コンティニュー回数、タイマー
    Score->Set(0);
    NumShips=3;
    NumContinues=0;
    Time=0;

    // 出現時の自機とレディ画面の生成
    MyShip=new CAppearingMyShip(0);
    new CReady();
}
```

## レディ画面

多くのゲームでは、タイトル画面でSTARTを押した瞬間にゲームが始まるのではなく、開始までに数秒間の待ち時間があります。この間にプレイヤーはゲームを始めるための心の準備をするわけです。

この待ち時間をどのように使うかはゲームによって異なりますが、一般には画面にメッセージを表示したり、自機の発進シーンを表示したりします。本書では、この画面を「レディ画面」と呼ぶことにします。ここでは画面に「GET READY」と表示し、同時に画面下から自機が出現するような演出を作ります（Fig. 7-4）。

**Fig. 7-4　レディ画面**

レディ画面は「GET READY」というメッセージを表示し、一定時間が経過したら、ゲームのステージを生成します。そして、自分は消滅します。

List 7-4は、レディ画面に相当するクラス（CReady）です。

**List 7-4　レディ画面（Scene.h、Scene.cpp）**

```
//「GET READY」と表示するレディ画面のクラス
// 画面の基本クラス（CScene）から派生する
class CReady : public CScene {
protected:

    // タイマー
    int Time;

public:

    // コンストラクタ、移動、描画
    CReady();
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CReady::CReady()
:   Time(0)
{}

// 移動
bool CReady::Move() {

    // 一定時間が経過したらステージを生成し、レディ画面を消去する
    Time++;
    if (Time>50) {
        new CStage();
        return false;
    }
    return true;
}

// 描画
void CReady::Draw() {
    int h=Game->GetGraphics()->GetHeight();
    Game->Font->DrawText(
        "GET READY", h/2-9*8, h/2-16, ColWhite, ColShade);
}
```

# ステージ

ゲームのステージが行う主な仕事は、敵の生成です。サンプルでは、一定時間ごとに敵の種類を変えながら、ランダムな位置に敵を出現させます。

多くのシューティングゲームでは、あらかじめ用意されたシナリオにしたがって、敵を決まった順番で出現させます。このような処理をスクリプトを使って実現する方法については、Chapter 8で解説します（P. 237）。

ステージは、ポーズ画面に関する処理も行います。プレイ中にキーボードの「C」キーまたはジョイスティックのボタン「2」を入力したら、ポーズ画面を表示します。

ポーズ画面を表示するときには、前フレームの入力を調べて、前フレームにポーズ用のキーやボタンが入力されていなかったときだけ、ポーズ画面を生成します。前フレームの入力を調べるのは、タイトル画面のメニュー選択と同様に、ボタンが何度も押されたと誤認しないためです。

ポーズ中には敵の生成を止めなければなりません。そこで、ポーズ状態になったら、ステージは何も行わないようにします。ポーズ中でなければ、敵の生成を行います。ここでは時間とともに敵の種類を切り替えながら、3種類の敵をランダムな位置に生成します。

List 7-5は、ステージに相当するクラス（CStage）です。

**List 7-5 ステージ（Scene.h、Scene.cpp）**

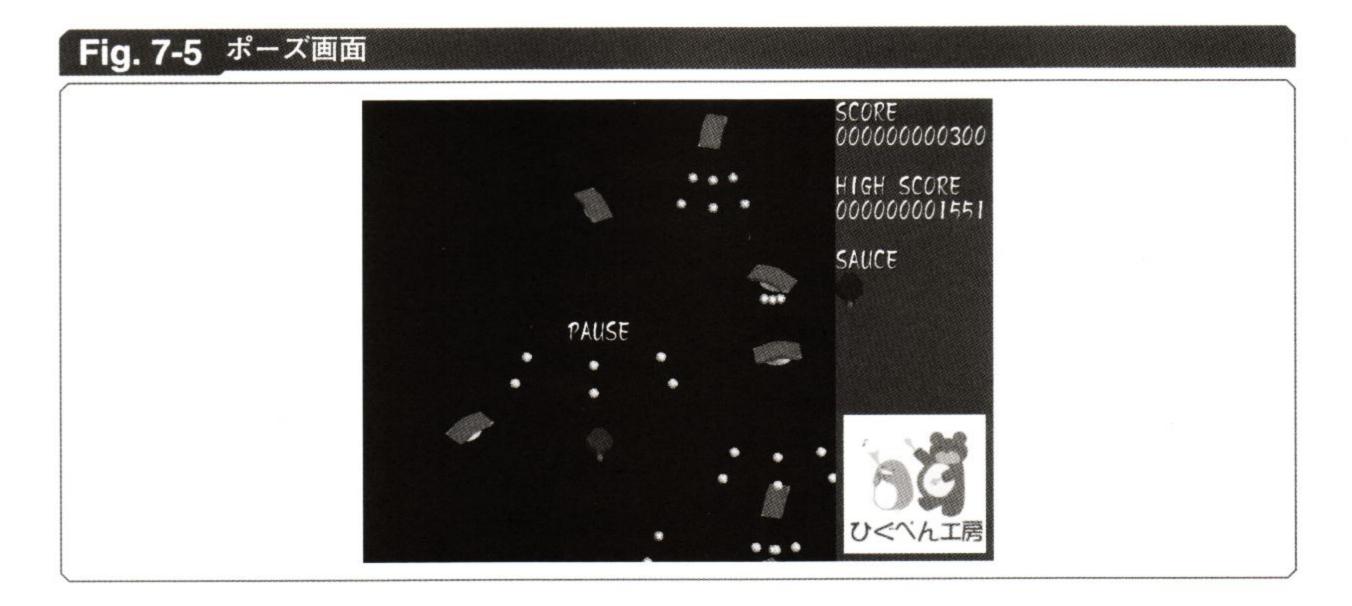
```
// ステージのクラス
// 画面の基本クラス（CScene）から派生する
class CStage : public CScene {
protected:

    // タイマー、前回の入力
    int Time;
    bool PrevInput;

public:

    // コンストラクタ、移動
    CStage();
    virtual bool Move();
};

// コンストラクタ
```

```
CStage::CStage()
:   Time(0), PrevInput(true)
{}

// 移動
bool CStage::Move() {

    // 特定のボタンが入力されたらポーズ画面を生成する
    const CInputState* is=Game->GetInput()->GetState(0);
    if (!Game->IsPaused() && !PrevInput && is->Button[2]) {
        new CPause();
    }

    // 前フレームの入力を調べるために
    // 入力をメンバ変数に保存しておく
    PrevInput=is->Button[2];

    // ポーズ中ならば何もせずに戻る
    if (Game->IsPaused()) return true;

    // 敵の生成
    switch (Time/600) {
        case 0:
            if (Time%40==0) new CAkami(Game->Rand05()*80);
            break;
        case 1:
            if (Time%40==0) new CTamago(Game->Rand05()*80);
            break;
        case 2:
            if (Time%40==0) new CEbi(Game->Rand05()*80);
            break;
        default:
            Time=0;
            break;
    };
    Time++;

    return true;
}
```

# ポーズ画面

ほとんどのゲームでは、ゲーム中にスタートボタンなどの特定のボタンを押すと、画面に「PAUSE」などと表示されて一時停止状態になります。この状態を「ポーズ画面」と呼ぶことにします。本書のサンプルでは、ゲーム中にキーボードの「C」キーまたはジョイスティックのボタン「2」を入力すると、ポーズ画面が表示されます (Fig. 7-5)。

ポーズ画面が表示されている間は、ゲームの進行を一時的に止める必要があります。これはタスクの動作を止めることによって実現します。

Fig. 7-5 ポーズ画面

## ゲームの進行を止める

ゲームの進行を止めるには、ゲーム本体の処理を拡張する必要があります。まず、ゲームの進行を止めるかどうかを表すフラグを用意します。進行を止めたいときには、このフラグをtrueにします。

次に、ゲームの移動処理を変更します。通常はすべてのタスクを動作させますが、フラグがtrueのときには、画面関係のタスク (画面の基本クラスから派生したクラスのインスタンス) だけを動作させます。これはポーズ画面のタスクを動作させるためです。ポーズの解除はポーズ画面が行うので、ポーズ画面のタスクも止めてしまうと、ポーズを解除することができなくなってしまいます。

画面関係のタスクを動作させると、ステージのタスクも動作します。しかし、ステージはポーズ中は何もしないようになっているので (P. 225)、ゲームの進行は止まります。

List 7-6は、ゲームの進行を止めるためのプログラムです。

**List 7-6 ゲームの進行を止める（Main.h、Main.cpp）**

```
class CShtGame : public CGame {

    // ...（中略）...

    // ゲームの進行を止めるためのフラグ
    bool Paused;

    // フラグに値を設定する関数
    void SetPaused(bool paused) {
        Paused=paused;
    }

    // フラグの値を取得する関数
    bool IsPaused() {
        return Paused;
    }
}

// ゲームの進行
void CShtGame::Move() {

    // ...（中略）...

    // タスクの動作
    if (!Paused) {
        MoveTask(BulletList);
        MoveTask(EnemyList);
        MoveTask(BeamList);
        MoveTask(EffectList);
        MoveTask(MyShipList);
        MoveTask(ShotList);
        Time++;
    }
    MoveTask(SceneList);

    // ...（中略）...

}
```

## ポーズ画面の表示

ポーズ画面を生成したら、前述のゲームの進行を止めるフラグをtrueにします。これで、ポーズ画面の表示と同時にゲームをポーズ状態にすることができます。

逆にポーズ画面を消去したら、ゲームの進行を止めるフラグをfalseにします。すると、ポーズ画面から抜けるのと同時に、ゲームの進行を再開することができます。

ポーズ画面は画面に「PAUSE」と表示します。そして、「C」キーまたはボタン「2」が入力されたら、ポーズを解除します。

List 7-7は、ポーズ画面に相当するクラス(CPause)です。ポーズ状態の設定と解除は、それぞれコンストラクタとデストラクタで行います。

**List 7-7 ポーズ画面(Scene.h、Scene.cpp)**

```
// ポーズ画面のクラス
// 画面の基本クラス(CScene)から派生する
class CPause : public CScene {
protected:

    // 前回の入力
    bool PrevInput;

public:

    // コンストラクタ、デストラクタ、移動、描画
    // ポーズ画面クラスは画面の基本クラスから派生しているので
    // デストラクタを正しく呼び出すためには
    // デストラクタを仮想関数にする必要がある
    CPause();
    virtual ~CPause();
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CPause::CPause()
:   PrevInput(true)
{
    // ポーズ状態にする
    Game->SetPaused(true);
}

// デストラクタ
CPause::~CPause()
```

```
{
    // ポーズ状態を解除する
    Game->SetPaused(false);
}

// 移動
bool CPause::Move()
{
    // 特定のキーやボタンが入力されたらポーズを解除する
    const CInputState* is=Game->GetInput()->GetState(0);
    if (!PrevInput && is->Button[2]) return false;
    PrevInput=is->Button[2];
    return true;
}

// 描画
void CPause::Draw() {
    int h=Game->GetGraphics()->GetHeight();
    Game->Font->DrawText(
        "PAUSE", h/2-5*8, h/2-16, ColWhite, ColShade);
}
```

## 残機を減らす

自機が弾や敵などに接触して爆発したら、残機を減らします。残機を減らすには、例えば自機の爆発を表すタスクに処理を追加するとよいでしょう。自機の爆発は一定時間で消滅しますが、このときに残機を-1します。

減らした後の残機が1以上ならば、爆発と同じ位置に新しい自機を復活させます。通常の自機とは違って、無敵期間を持った復活時の自機を出現させます。一方、残機が0の場合にはコンティニュー画面(P. 228)を表示します。

List 7-8は、残機を減らすプログラムです。

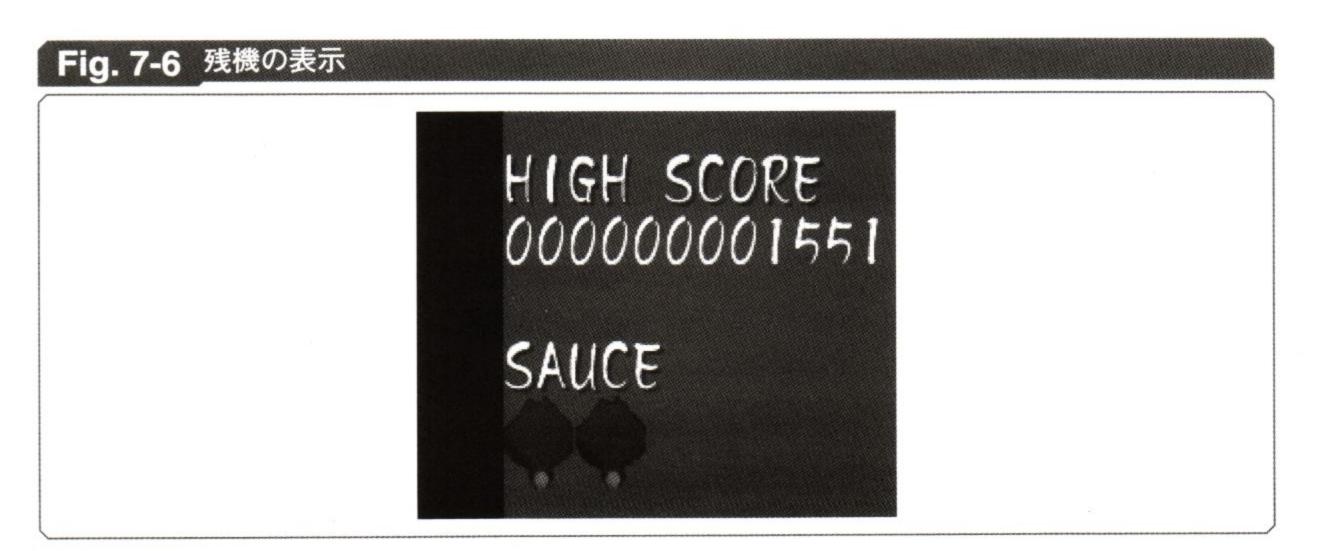
**List 7-8 残機を減らすプログラム (Effect.cpp)**

```
// 自機の爆発を表すクラスの移動処理
bool CMyShipCrash::Move() {
    Time++;
    if (Time>50) {

        // 残機を減らす
```

```
            // DecNumShipsは残機を減らす関数（Main.h参照）
            Game->DecNumShips();

            // 残機があれば自機を復活させる
            if (Game->GetNumShips()>0) {
                Game->SetMyShip(new CRevivalMyShip(X, Y));
            }

            // 残機がなければコンティニュー画面を表示する
            else {
                Game->SetMyShip(NULL);
                new CContinue();
            }

            // 爆発を消す
            return false;
        }
        return true;
    }
```

## 残機の表示

残機数はスコア領域に表示することにします。ここでは残機数に応じて、自機の2D画像を表示します（Fig. 7-6）。2D画像の表示には、Chapter 2のテクスチャクラス（P. 32）を使用します。

Fig. 7-6 残機の表示

List 7-9は、残機数を表示するプログラムです。ゲーム本体クラス（CShtGame）の描画処理（Draw関数）に処理を追加しました。

**List 7-9　残機の表示（Main.cpp）**

```
void CShtGame::Draw()
{
    // ...（中略）...

    // スコア領域の初期化
    viewport.X=h;
    viewport.Width=w-h;
    device->SetViewport(&viewport);
    Graphics->Clear(D3DCOLOR_XRGB(100, 50, 80));

    // スコア、ハイスコアの描画
    Font->DrawText("SCORE", h, 0, ColWhite, ColShade);
    Font->DrawText(Score->GetDigits(), h, 32, ColWhite);
    Font->DrawText("HIGH SCORE", h, 96, ColWhite, ColShade);
    Font->DrawText(HighScore->GetDigits(), h, 128, ColWhite);

    // 残機数の表示
    Font->DrawText("SAUCE", h, 192, ColWhite, ColShade);
    for (int i=0; i<NumShips-1; i++) {
        MyShipIcon->Draw(h+i*36, 224, ColWhite);
    }

    // ...（中略）...

}
```

## コンティニュー画面

残機が0になった場合には、コンティニュー画面を表示します（Fig. 7-7）。今回は、コンティニュー画面で「C」キーまたはボタン「2」を押したときに、ゲームを続行できるようにしました。

コンティニュー画面には「CONTINUE?」というメッセージとともに、カウントが表示されます。カウントは約1秒ごとに1ずつ減っていき、0になるとゲームオーバー画面に移行します。「X」キーまたはボタン「1」を押すと、カウントを進めることができます。

コンティニュー画面の生成時には、ゲームの進行を止めます。逆にコンティニュー画面

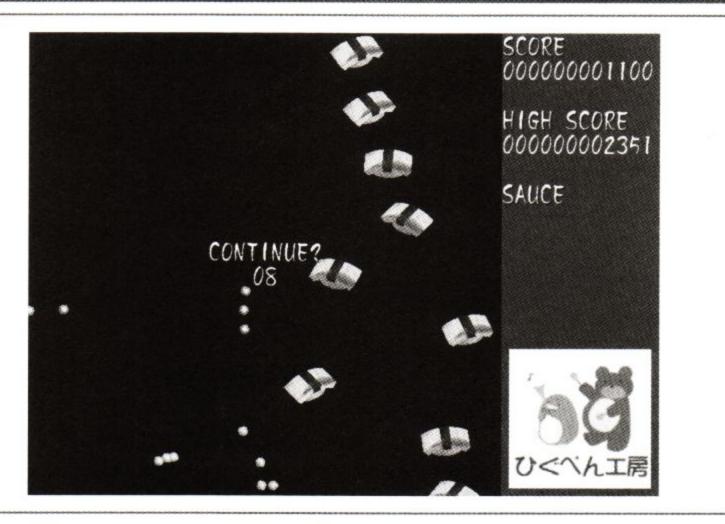
の消去時には、ゲームの進行を再開します。これはポーズ画面の処理と同様です。

コンティニュー画面の表示中には、以下の処理を行います。

- 「C」キーまたはボタン「2」が押されたらコンティニューする
- 「X」キーまたはボタン「1」が押されたらカウントを減らす
- 約1秒ごとにカウントを減らす
- カウントが0になったらゲームオーバー画面を表示する

コンティニューする場合には、スコア・残機・タイマーを初期化して、ゲームを再開します。また、コンティニューをしたときには、スコアの下1桁をコンティニュー回数に等しくします。これはスコアからコンティニュー回数がわかるようにするためです。この手法は多くのゲームで使われています。

List 7-10は、コンティニュー画面に相当するクラス（CContinue）です。

Fig. 7-7 コンティニュー画面

**List 7-10 コンティニュー画面（Scene.h、Scene.cpp、Main.cpp）**

```
// コンティニュー画面のクラス
// 画面の基本クラス（CScene）から派生する
class CContinue : public CScene {
protected:

    // タイマー、カウント、前回の入力
    int Time, Count;
    bool PrevInput;

public:
```

```
    // コンストラクタ、デストラクタ、移動、描画
    CContinue();
    virtual ~CContinue();
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CContinue::CContinue()
:   Time(0), Count(10), PrevInput(true)
{
    // ポーズ状態にする
    Game->SetPaused(true);
}

// デストラクタ
CContinue::~CContinue()
{
    // ポーズ状態を解除する
    Game->SetPaused(false);
}

// 移動
bool CContinue::Move() {

    // ボタン入力
    const CInputState* is=Game->GetInput()->GetState(0);
    if (!PrevInput) {

        // コンティニューする
        if (is->Button[2]) {
            Game->Continue();
            Game->SetMyShip(new CRevivalMyShip(0, 30));
            return false;
        }

        // カウントを減らす
        if (is->Button[1]) {
            if (Count>0) {
                Count--;
                Time=0;
            }
        }
    }
    PrevInput=is->Button[0]||is->Button[1];
```

```
    // カウントダウン
    Time++;
    if (Time>60) {
        Time=0;
        if (Count==0) {
            new CGameOver();
            return false;
        }
        Count--;
    }
    return true;
}

// 描画
void CContinue::Draw() {
    int h=Game->GetGraphics()->GetHeight();
    char buf[]={'0'+Count/10, '0'+Count%10, '¥0'};
    Game->Font->DrawText(
        "CONTINUE?", h/2-9*8, h/2-32, ColWhite, ColShade);
    Game->Font->DrawText(
        buf, h/2-2*8, h/2+0, ColWhite, ColShade);
}

// コンティニューの処理
void CShtGame::Continue() {

    // スコアの下1桁をコンティニュー回数にして初期化する
    if (NumContinues<9) NumContinues++;
    Score->Set(NumContinues);

    // 残機とタイマーの初期化
    NumShips=3;
    Time=0;
}
```

# ゲームオーバー画面

コンティニューをしなかった場合には、ゲームオーバー画面を表示します（Fig. 7-8）。そして、一定時間が経過したらタイトル画面へ戻ります。

ゲームオーバー画面では「GAME OVER」と表示し、ハイスコアだった場合にはさらに「YOU'VE GOT THE BEST SCORE!」と表示します。そして一定時間が経過したら、後述するゲーム終了処理（P. 234）を呼び出して、ゲームを終了します。

List 7-11は、ゲームオーバー画面に相当するクラス（CGameOver）です。

**Fig. 7-8** ゲームオーバー画面

**List 7-11** ゲームオーバー画面（Scene.h、Scene.cpp、Main.h）

```
// ゲームオーバー画面のクラス
// 画面の基本クラス（CScene）から派生する
class CGameOver : public CScene {
protected:

    // タイマー
    int Time;

    // ハイスコアかどうか
    bool IsHighScore;

public:

    // コンストラクタ、移動、描画
```

```
    CGameOver();
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CGameOver::CGameOver()
:   Time(0)
{
    // ハイスコアかどうかを調べる
    IsHighScore=Game->IsHighScore();
}

// 移動
bool CGameOver::Move()
{
    // 一定時間後にタイトル画面へ移行する
    Time++;
    if (Time>150) {
        Game->End();
        return false;
    }
    return true;
}

// 描画
void CGameOver::Draw() {
    int h=Game->GetGraphics()->GetHeight();

    // ハイスコアの場合の描画
    if (IsHighScore) {
        Game->Font->DrawText(
            "GAME OVER", h/2-9*8, h/2-32, ColWhite, ColShade);
        Game->Font->DrawText(
            "YOU'VE GOT THE BEST SCORE!", h/2-24*8, h/2+0,
            ColWhite, ColShade);
    }

    // ハイスコアではない場合の描画
    else {
        Game->Font->DrawText(
            "GAME OVER", h/2-9*8, h/2-16, ColWhite, ColShade);
    }
}

// ハイスコアかどうかを調べる
```

```
bool CShtGame::IsHighScore() {
    return Score->Compare(HighScore)==0;
}
```

## ゲームの終了処理

ゲームの終了時には、全タスクを消去した後に、タイトル画面を生成します。これでゲームを起動したときの状態に戻ります。

List 7-12は、ゲームの終了処理に関するプログラムです。この終了処理はゲーム本体クラス（CShtGame）の移動処理（Move関数）に追加しました。

**List 7-12　ゲームの終了処理（Main.h、Main.cpp）**

```
class CShtGame : public CGame {

    // ...（中略）...

    // ゲーム終了のフラグ
    bool GameEnd;

    // ゲーム終了のフラグを設定する関数
    // ゲームオーバー画面を消去するときに呼び出す（List 7-11）
    void End() {
        GameEnd=true;
    }
}

// ゲーム全体の動作
void CShtGame::Move() {

    // ...（中略）...

    // タスクの動作
    if (!Paused) {
        MoveTask(BulletList);
        MoveTask(EnemyList);
        MoveTask(BeamList);
        MoveTask(EffectList);
        MoveTask(MyShipList);
        MoveTask(ShotList);
        Time++;
    }
    MoveTask(SceneList);
```

```
    // ゲーム終了処理
    if (GameEnd) {

        // 全タスクの消去
        BeamList->DeleteTask();
        BulletList->DeleteTask();
        EffectList->DeleteTask();
        EnemyList->DeleteTask();
        MyShipList->DeleteTask();
        ShotList->DeleteTask();
        SceneList->DeleteTask();

        // タイトル画面の生成
        new CTitle();

        // フラグの解除
        GameEnd=false;
    }
}
```

## >>Chapter 7のまとめ

本章ではゲームの外枠となる部分に注目して、タイトル画面、レディ画面、ポーズ画面、残機の管理、コンティニュー画面、ゲームオーバー画面について解説しました。今まではプログラムを起動するといきなりゲームが始まっていましたが、タイトル画面などを追加したことによって、ゲームとしての体裁が整いました。

次章ではステージをテーマに、背景の表示や敵の出現シーケンスなどについて説明します。

# Chapter 08

# ステージ

自機・弾・敵といったゲームに必要な要素がそろい、タイトル画面やゲームオーバー画面などの外枠も用意しました。そこで本章では、より本格的で面白いゲームに仕上げるため、ステージに手を加えます。まずはステージの背景を表示し、次にステージ展開を盛り上げるいろいろな敵を作ります。さらに、ステージの進行をスクリプトで管理する方法について説明します。

Shooting Game Programming

# 背景、ステージ進行、BGMの追加

本章では、サンプルに対して、

- 背景
- 大きな敵
- ステージ進行
- BGM

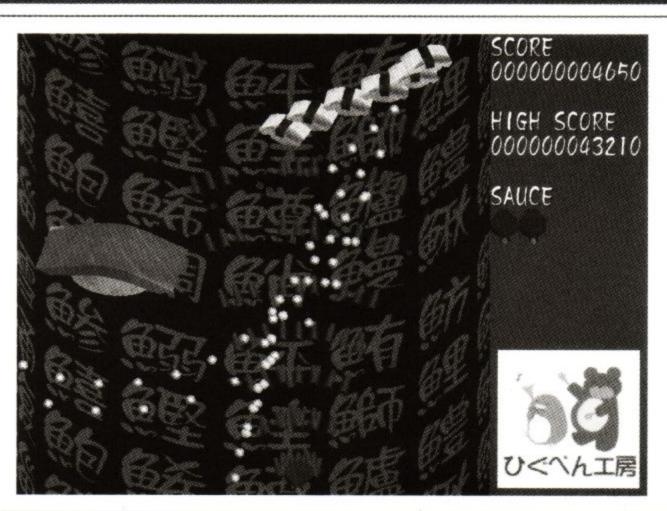
などを追加します。プロジェクトは付録CD-ROMの「ShtGame_Stage」フォルダに収録しました。実行ファイルは「ShtGame_Stage¥Release¥ShtGame.exe」です。

Fig. 8-1はサンプルの実行画面です。

BGMに関しては、Kohzas氏のご厚意により、Webサイト「Kohzas Station」に掲載されている数々の素晴らしい曲のいくつかを使わせていただきました。篤く御礼申し上げます。

**Kohzas Station**
http://www.kohzas.net/

Kohzas Stationではゲームなどで自由に使用できる楽曲が数多く提供されています。ゲームを作りたいけれども曲を作るのが大変！　という方には大変お勧めのサイトですので、ぜひ一度ご覧になってみてください。

**Fig. 8-1** 背景・大きな敵・ステージ進行・BGMなどを追加したサンプル

# 背景

昔は背景がないシューティングゲームもありましたが、現在はほとんどのゲームが背景を持っています。最近の縦スクロールシューティングゲームの場合、多くのゲームでは背景に当たり判定がないため、仮に背景がなくてもゲームの本質はさほど変わりません。しかし、やはり背景があった方が見た目はずっとにぎやかになります。

本書のサンプルにも背景を追加してみました（Fig. 8-2)。3Dグラフィックを使って奥行き感のある背景を描画しています。

ほとんどの縦スクロールゲームでは、時間とともに画面の上から下に向かって背景が流れます。このスクロールで、自機がマップ上を進んでいく雰囲気を演出するのです。

また、多くのゲームでは、自機を左右に動かすと背景がそれとは逆の方向に動きます。この左右方向への動きがあると、画面を広く感じさせることができます。本書のサンプルでは背景に円筒を表示していますが、自機を左右に動かしたときに、この円筒を移動とは逆の方向に少し回転させています。

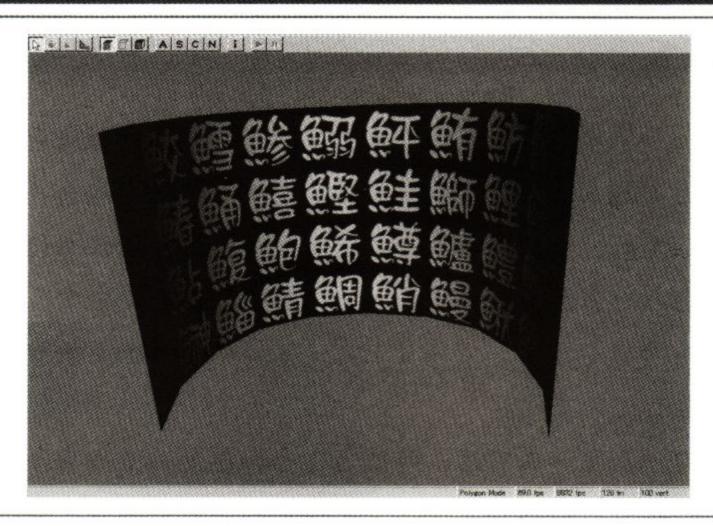
背景を移動して描画する処理も、自機や弾と同様に、フレームごとに呼び出します。そして、タイマーを使って背景の上下位置を計算し、背景を下方向にスクロールさせます。また、自機の左右位置から背景の左右位置（回転角度）を計算して、背景を左右方向に動かします。

本書のサンプルでは、背景用の3Dオブジェクトを縦に並べて描画することによって、スクロールする背景を表現します。3DオブジェクトはFig. 8-3のようなものです。

List 8-1は、背景に関する処理をまとめたクラス（CBGSakanaTube）です。

**Fig. 8-2** 背景

SCORE
000000000000
HIGH SCORE
000000043210
SAUCE
ひぐぺん工房

**Fig. 8-3 背景用の3Dオブジェクト**

**List 8-1 背景の表示（CStage.h、CStage.cpp）**

```
// 背景のクラス
class CBGSakanaTube : public CMover {
protected:

    // タイマー
    // スクロールの位置を計算するために使用
    int Time;

public:

    // new演算子、delete演算子
    // 引数のBackListは背景タスクリスト
    void* operator new(size_t t) {
        return operator_new(t, Game->BackList);
    }
    void operator delete(void* p) {
        operator_delete(p, Game->BackList);
    }

    // コンストラクタ
    // タイマーを初期化
    CBGSakanaTube()
    :   CMover(Game->BackList, 0, 0, 0), Time(0)
    {}

    // 移動および描画
    virtual bool Move();
    virtual void Draw();
```

```
};

// 移動
bool CBGSakanaTube::Move() {

    // タイマーの更新
    // 背景の表示位置計算は描画処理で行う
    if (!Game->IsPaused()) Time++;

    return true;
}

// 描画
void CBGSakanaTube::Draw() {

    // 時間とともに背景をスクロールさせる
    float y=-(float)(Time%120);

    // 自機の位置に応じて背景の回転角度を決める
    CMyShip* myship=Game->GetMyShip();
    float turn=myship?-myship->X/400:0;

    // 複数の背景用3Dオブジェクトを並べて描画する
    for (int i=0; i<4; i++, y+=60) {
        Game->MeshSakanaTube->Draw(
            0, BACK_Z, y, 40, 40, 40,
            0.125f, 0, turn, TO_ZX, 1, false);
    }
}
```

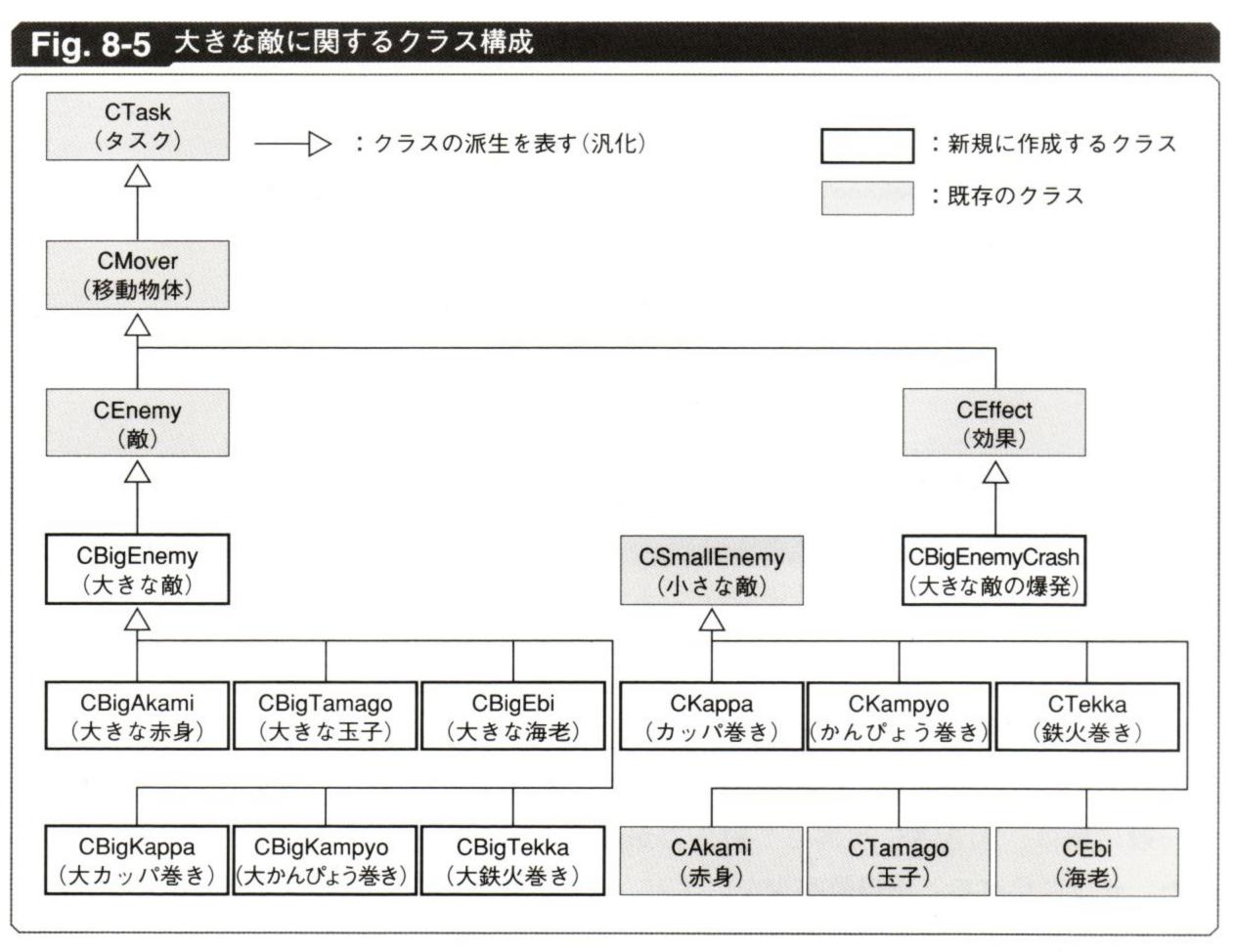
## 大きな敵でステージに緩急をつける

多くのシューティングゲームには、大きさが異なる何種類かの敵が出現し、ステージのリズムを形作っています。ザコと呼ばれる小さな敵、中型機などと呼ばれるザコよりも大きめの敵、そしてボスと呼ばれる非常に大きな敵がいます。ここではザコよりもやや大きめの敵を作る方法を解説します。ボスについてはChapter 9で解説します(P. 277)。

一般に敵は、大きさに比例して耐久力が高く(つまり硬く)、スコアも高くなります。また、ザコに比べると出現数が少ないぶん、激しい攻撃をします。つまり、ザコよりもずっと多くの弾を出します(Fig. 8-4)。

本書では大きな敵の共通処理をCBigEnemyというクラスにまとめました（Fig. 8-5）。各クラスの役割は次のとおりです。

**Fig. 8-4 大きな敵**

**Fig. 8-5 大きな敵に関するクラス構成**

### CBigEnemy

大きな敵の共通処理をまとめたクラスです。CEnemyクラスから派生します。あらゆる大きな敵のクラスは、このCBigEnemyクラスから派生します。

### CBigAkami

大きな赤身のクラスです。CBigEnemyクラスから派生します。

### CBigTamago

大きな玉子のクラスです。CBigEnemyクラスから派生します。

### CBigEbi

大きな海老のクラスです。CBigEnemyクラスから派生します。

### CBigKappa

大きなカッパ巻きのクラスです。CBigEnemyクラスから派生します。

### CBigKampyo

大きなかんぴょう巻きのクラスです。CBigEnemyクラスから派生します。

### CBigTekka

大きな鉄火巻きのクラスです。CBigEnemyクラスから派生します。

### CKappa

小さなカッパ巻きのクラスです。CEnemyクラスから派生します。

### CKampyo

小さなかんぴょう巻きのクラスです。CEnemyクラスから派生します。

### CTekka

小さな鉄火巻きのクラスです。CEnemyクラスから派生します。

### CBigEnemyCrash

大きな敵の爆発を表示するためのクラスです。大きな敵が爆発したときに、画面上の弾を消す処理も行います。CEffectクラスから派生します。

## 迫力を出すには

大きな敵の作り方は、実はザコ（小さな敵）の作り方とあまり変わりません。ただし、強く迫力のある敵になるようにパラメータを設定し、演出方法も派手なものを使います。例えば、耐久力とスコアは小さな敵よりも高くします。キャラクターの大きさが大きくなったぶん、当たり判定も大きめにします。

また、上方の画面外から出現するように、出現位置を調整します。敵が大きいので、小さな敵よりも出現位置は画面枠から離して設定する必要があります。

小さな敵と同様に、耐久力が0以下になったら大きな敵は爆発します。大きな敵が爆発したときには、爆発も大きなものを表示するとよいでしょう。

List 8-2は、大きな敵の共通処理をまとめたCBigEnemyクラスです。この例で使用する3Dモデルは小さな敵と共通ですが、拡大して描画することにより、大きな敵を表現しています。もちろん、大きい敵専用の3Dモデルを使用することもできます。

**List 8-2　大きな敵の共通処理（Enemy.h、Enemy.cpp）**

```
// 大きな敵
class CBigEnemy : public CEnemy {
public:

    // コンストラクタ、移動、描画
    CBigEnemy(CMesh* mesh, float x);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CBigEnemy::CBigEnemy(CMesh* mesh, float x)
:   CEnemy(mesh, x, -62, -7.5f, -7.5f, 7.5f, 7.5f, 100, 1000)
{}

// 移動
bool CBigEnemy::Move() {

    // 耐久力が0以下になった場合
    if (Vit<=0) {

        // スコアの加算
        Game->AddScore(Score);

        // 爆発の表示
```

```
        new CBigEnemyCrash(X, Y);

        // タスクの消去
        return false;
    }

    // 画面下方に出たらタスクを消去
    return Y<50+24;
}

// 描画
// 第4引数～第6引数の3, 3, 3が拡大率を表す
void CBigEnemy::Draw() {
    DrawMesh(X, Z, -Y, 3, 3, 3,
        0.125f, Yaw, 0, TO_ZYX, 1, false);
}
```

## 大きな敵の例

大きな敵の例として、大きな赤身のにぎりの姿をした敵を作りました。大きな赤身は小さな赤身よりもやや動きが遅く、激しい攻撃をします。

List 8-3は、大きな赤身のクラス（CBigAkami）です。移動処理（CBigAkami::Move関数）を、小さな赤身の移動処理（CAkami::Move関数）と見比べてみてください（P. 193）。

**List 8-3　大きな敵（Enemy.h、Enemy.cpp）**

```
// 大きな赤身
class CBigAkami : public CBigEnemy {
public:

    // コンストラクタ、移動
    CBigAkami(float x);
    virtual bool Move();

    // スクリプト機能（P. 256）で使用する関数
    static CEnemy* New(float x) { return new CBigAkami(x); }
};

// コンストラクタ
// CBigEnemyクラスのコンストラクタを呼び出す
// 引数は3DモデルとX座標
CBigAkami::CBigAkami(float x)
:   CBigEnemy(Game->MeshAkami, x)
```

```
{}

// 移動
bool CBigAkami::Move() {
    CMyShip* myship=Game->GetMyShip();

    // 座標、回転角度、タイマーの更新
    Y+=0.4f;
    Yaw+=0.01f;
    Time++;

    // 弾を扇状にばらまく（n-way弾）
    if (myship && Time%10==0 && Time%100<20) {
        if (Time%200<100) {
            for (int i=-8; i<=8; i++) {
                new CDirBullet(Game->MeshBullet,
                    Color, X, Y, 0.25f+i*0.03f, 1.0f, 0);
            }
        } else {
            for (int i=-7; i<=8; i++) {
                new CDirBullet(Game->MeshBullet,
                    Color, X, Y, 0.23f+i*0.03f, 1.0f, 0);
            }
        }
    }

    // 大きな敵の共通処理
    return CBigEnemy::Move();
}
```

## 大きな敵を破壊したときに弾を消す

ゲームによっては、大きな敵を破壊したときに画面上の弾が消えることがあります(Fig. 8-6)。回避が難しくなるほどの弾を撃たれてしまっても、敵を破壊すれば弾が消せるとなれば、プレイヤーは大きな敵を破壊するために積極的なプレイをすることになります。そうすれば、ゲームの緊張感をいい具合に増すことができるというわけです。

本書のサンプルでは、画面上のすべての弾を消すのではなく、破壊された大きな敵の周囲にある弾だけを消すようにしました。このルールにおいては、例えば、大きな敵の周囲に弾を誘導してから破壊して効果的に弾を消そう、といった戦術が考えられます。すべての弾を消す場合に比べて、弾の誘導に工夫が必要なぶん、面白いゲームになりそうです。

弾を消す処理は、爆発タスクの生成時に行うことにします。すべての弾に関して、爆発の中心から弾までの距離を調べます。そして、一定距離内にある弾だけを消去します。

なお、大きな敵を破壊したときに弾を消すだけではなく、得点アイテムなどを出現させることも可能です。アイテムの出現についてはChapter 10で解説します（P. 305）。

List 8-4は、大きな敵の爆発を行うクラス（CBigEnemyCrash）です。大きな敵を破壊したときに画面上の弾を消す処理は、このクラスが行います。弾を消去するCBullet::Crash関数は、次のList 8-5に掲載しました。

**Fig. 8-6 大きな敵を破壊すると弾が消える**

**List 8-4 大きな敵の爆発（Effect.h、Effect.cpp）**

```
// 大きな敵の爆発
class CBigEnemyCrash : public CEnemyCrash {
public:

    // コンストラクタ
    CBigEnemyCrash(float x, float y);
};

// コンストラクタ
CBigEnemyCrash::CBigEnemyCrash(float x, float y)
:   CEnemyCrash(x, y, 0.6f)
{
    // 弾の消去
    for (CTaskIter i(Game->BulletList); i.HasNext(); ) {
        CBullet* bullet=(CBullet*)i.Next();

        // 爆発の中心から一定距離内にある弾だけを消去する
        float dx=x-bullet->X, dy=y-bullet->Y;
        if (dx*dx+dy*dy<1000) {
```

```
                // 弾を消去する（List 8-5）
                bullet->Crash();
                i.Remove();
            }
        }

        // 効果音の再生
        Game->PlaySE(Game->SEBigCrash);
}
```

## 弾が消えるときのエフェクト

大きな敵の破壊によって弾が消えたときにエフェクトを表示すると、破壊の成功をプレイヤーに印象づけることができます。タスクシステムを使う場合には、例えば、弾のタスクを消去して同じ位置に弾を消去するエフェクトのタスクを生成すれば、エフェクトが表示できます。

List 8-5は、弾を消去するエフェクトに関するプログラムです。

**List 8-5　弾の消去（Bullet.cpp、Effect.h、Effect.cpp）**

```
// 弾の消去エフェクトを生成
void CBullet::Crash() {
    new CBulletCrash(X, Y, Mesh);
}

// 弾を消去するエフェクト
class CBulletCrash : public CEffect {
protected:

    // 3Dモデル、時間
    CMesh* Mesh;
    int Time;

public:

    // コンストラクタ、移動、描画
    CBulletCrash(float x, float y, CMesh* mesh);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CBulletCrash::CBulletCrash(float x, float y, CMesh* mesh)
```

```
:   CEffect(x, y), Mesh(mesh), Time(0)
{}

// 移動
// 一定時間が経過したらfalseを返す
bool CBulletCrash::Move() {
    Time++;
    return Time<=10;
}

// 描画
void CBulletCrash::Draw() {
    float
        time=(float)Time/10,

    // 時間とともに拡大率と角度を変化させる
    sx=1-time,
    sy=1+time,
    yaw=time/2;
    Mesh->Draw(
        X, Z, -Y, sx, 1, sy,
        0, yaw, 0, TO_Y, 1, false);
}
```

## 敵の編隊

同じ位置に少しずつタイミングをずらして同一の敵を生成すると、敵が1列の編隊を組んで飛んでくるような状態が作れます。出現位置や出現タイミングを調整すれば、さまざまな形の編隊を作ることが可能です。

後述するスクリプトのシステムを使えば、簡単に敵の編隊を出現させることができます(P. 256)。本章のサンプルでは、さまざまな陣形の敵と実際に戦うことができるので、ぜひ一度遊んでみてください。

一方で、単に並んで飛んでくるというだけではなく、より緊密に連携して複雑な動きをする編隊を作ることもできます。本章では以下のような敵を作りました。

### カッパ巻き

大きな敵の周囲を小さな敵が腕のように回りつつ、弾を発射します (Fig. 8-7)。

### かんぴょう巻き

大きな敵の周囲を小さな敵が守りつつ、弾を発射します（Fig. 8-8）。

### 鉄火巻き

大きな敵から小さな敵が触手となって生えていて、自機を攻撃します（Fig. 8-9）。

※

これらの編隊に関しては、大きな敵の周囲にいる小さな敵は破壊することができません。中心の大きな敵を破壊すれば、小さな敵ごと破壊することができます。

こういった緊密に連携して動く敵を作るには、出現位置や出現タイミングを調整するだけではなく、敵のクラス間で連携して処理を進める必要があります。

**Fig. 8-7** カッパ巻き

**Fig. 8-8** かんぴょう巻き

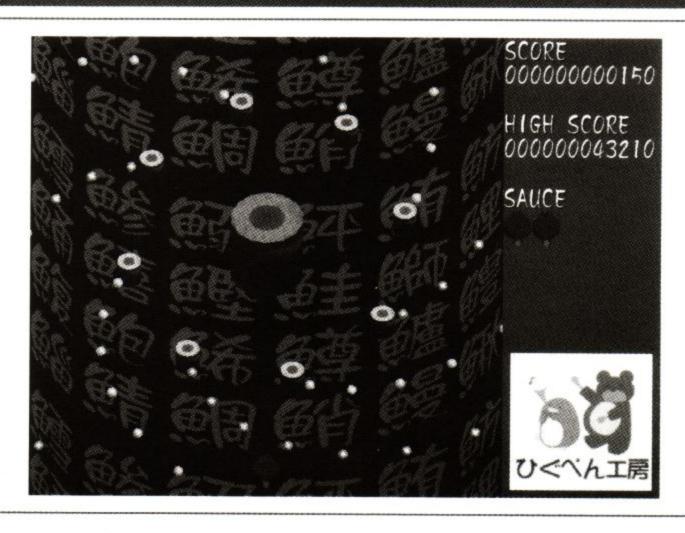



Fig. 8-9 鉄火巻き

## 連携する小さな敵

ここではカッパ巻きを例に、連携して動く敵の作り方を解説します。本書のサンプルの場合、連携して動く敵は小さな敵と大きな敵に分かれています。まずは小さなカッパ巻きについて説明します。

小さなカッパ巻きは破壊することはできませんが、当たり判定は持っていて、当たった攻撃はそこで消えてしまいます。そのため、小さなカッパ巻きの間からショットやビームを打ち込んで、大きなカッパ巻きを倒すことになります。このとき、小さなカッパ巻きの当たり判定が大きいと、大きな敵の破壊が難しくなってしまうので、当たり判定を小さめにしました。このように、状況に応じて当たり判定の大きさを調整して、楽しく遊べるバランスを探していくのが、面白いゲームを作るためのポイントになります。

小さなカッパ巻きは、大きなカッパ巻きと連携するために、大きなカッパ巻きの状態を調べて自らの行動を決定します。大きなカッパ巻きが破壊された場合には、小さなカッパ巻きも破壊して、スコアを加算します。大きなカッパ巻きが画面外に出た場合には、小さなカッパ巻きも単に消去します。

小さなカッパ巻きは、大きなカッパ巻きの周囲を回るように動きます。この動きのために、大きなカッパ巻きに対する回転角度と、大きなカッパ巻きの中心からの距離を保持しておきます。そして、フレームごとに回転角度を変化させて座標を計算します。

List 8-6は、小さなカッパ巻きのプログラムです。

**List 8-6 連携して動く小さな敵のプログラム（Enemy.h、Enemy.cpp）**

```
// 小さなカッパ巻き
class CKappa : public CSmallEnemy {

    // 大きなカッパ巻きへのポインタ
    CBigKappa* Parent;

    // 回転角度、中心からの距離、タイマー
    float Rad, Dist;
    int Time;

public:

    // コンストラクタ、移動
    CKappa(CBigKappa* parent, float rad, float dist);
    virtual bool Move();
};

// コンストラクタ
CKappa::CKappa(CBigKappa* parent, float rad, float dist)
:   CSmallEnemy(Game->MeshKappa, 0),
    Parent(parent), Rad(rad), Dist(dist), Time(0)
{
    // L、T、R、Bは当たり判定の左、上、右、下の相対座標
    L=T=-1.5f;
    R=B=1.5f;
}

// 大きな敵の状態を表す定数
enum { ALIVE, DEAD, GONE };

// 移動
bool CKappa::Move() {
    CMyShip* myship=Game->GetMyShip();

    // 位置と角度の更新
    X=Parent->X+cosf(Rad)*Dist;
    Y=Parent->Y+sinf(Rad)*Dist;
    Rad+=0.04f;

    // 一定時間ごとに方向弾を発射する
    Time++;
    if (Time%10==0) {
        new CDirBullet(Game->MeshNeedle,
            Color, X, Y, Rad*(0.5f/D3DX_PI)-0.25f, 0.8f, 0);
    }
```

```
    // 大きな敵が破壊されたか画面外に出た場合の処理
    if (Parent->State!=ALIVE) {

        // 破壊されていたら小さな敵も破壊してスコアを加算
        if (Parent->State==DEAD) {
            Game->AddScore(Score);
            new CEnemyCrash(X, Y);
        }
        return false;
    }
    return true;
}
```

## 編隊のリーダー

大きなカッパ巻きは、いわばカッパ巻き編隊のリーダーです。しかし、この編隊の場合には、小さなカッパ巻きの方で連携のための機能を備えているため、リーダーのふるまいは通常の大きな敵とだいたい同じです。ただし、周囲に小さなカッパ巻きをしたがえているため、編隊全体の大きさはかなり大きくなります。出現時に編隊の一部が画面内に急に現れてしまわないよう、大きなカッパ巻きの出現位置を上ぎみに設定するとよいでしょう。

大きなカッパ巻きが画面外に移動したと判定されると、小さなカッパ巻きを含めた編隊全体が消去されます。編隊全体が画面外へ出たときにかぎって編隊が消去されるように、画面外に出たかどうかの判定をする際には、座標を適切に調整する必要があります。

カッパ巻き編隊の処理でポイントとなるのは、リーダーを消去するタイミングです。リーダーの破壊や画面外への移動に伴って、小さなカッパ巻きも消去されなければなりません。リーダーの状態が変わったことを小さなカッパ巻きが検知する必要があるため、状態を変えた直後にリーダーが消去されてしまっては不都合です。

そこで、状態が変わってもすぐにリーダーを消去せずに、1フレームの間はそのまま生き残るようにします。そして、次のフレームで再びリーダーの移動処理が呼び出されたところで、状態を判定してリーダーを消去します。すると、リーダーを消去する前に小さなカッパ巻きの移動処理（List 8-6のCKappa::Move関数）が必ず一度実行され、編隊としてまとまったふるまいを実現できます。

List 8-7は、大きなカッパ巻きのプログラムです。

**List 8-7 連携して動く大きな敵のプログラム（Enemy.h、Enemy.cpp）**

```
// 大きなカッパ巻き
class CBigKappa : public CBigEnemy {
public:

    // 状態
    int State;

    // コンストラクタ、移動、生成
    CBigKappa(float x);
    virtual bool Move();
    static CEnemy* New(float x) { return new CBigKappa(x); }
};

// コンストラクタ
// 状態をALIVE（活動中）に設定
CBigKappa::CBigKappa(float x)
:   CBigEnemy(Game->MeshKappa, x),
    State(ALIVE)
{
    // Y座標を少し上ぎみに設定する
    Y=-50-24;

    // 小さな敵を大きな敵の周囲に生成する
    for (int i=0, n=3, c=Color; i<n; i++) {
        for (int j=0; j<4; j++, c=1-c) {
            CKappa* k=new CKappa(this, i*D3DX_PI*2/n, 14+j*8);
            k->Color=c;
        }
    }
}

// 移動
bool CBigKappa::Move() {

    // すでに「破壊ずみ」か「画面外へ移動ずみ」の状態に
    // なっていたら消去する
    if (State!=ALIVE) return false;

    // 下方向に進む
    Y+=0.2f;

    // 耐久力が0以下ならば破壊
    if (Vit<=0) {
        // スコアを加算する
        Game->AddScore(Score);
```

```
        new CBigEnemyCrash(X, Y);

        // 状態をDEAD（破壊ずみ）にする
        State=DEAD;
    }

    // 画面外に出たら状態をGONE（画面外へ移動ずみ）にする
    // Y座標のしきい値を下ぎみにし、
    // 編隊全体が画面外に出たら状態GONEになるようにする
    if (Y>=50+24+14+8*4) State=GONE;

    // 消去すべき状態に遷移してもとりあえず1回trueを返す
    // これは、消去される前に仲間に自分の状態の変化を知らせるための工夫
    return true;
}
```

### 編隊のヴァリエーション

かんぴょう巻きと鉄火巻きについても、カッパ巻きとほぼ同じ方法で実現しています。詳細は付録CD-ROMの「Enemy.h」と「Enemy.cpp」をご覧ください。

連携して動く敵は、ボスに応用しても面白いでしょう。多くの触手を持った敵や、多関節を使って歩く敵なども、ここで紹介したのと同様の方法で実現することができます。ぜひ、いろいろとアイディアを凝らしてみてください。

## 敵の集団

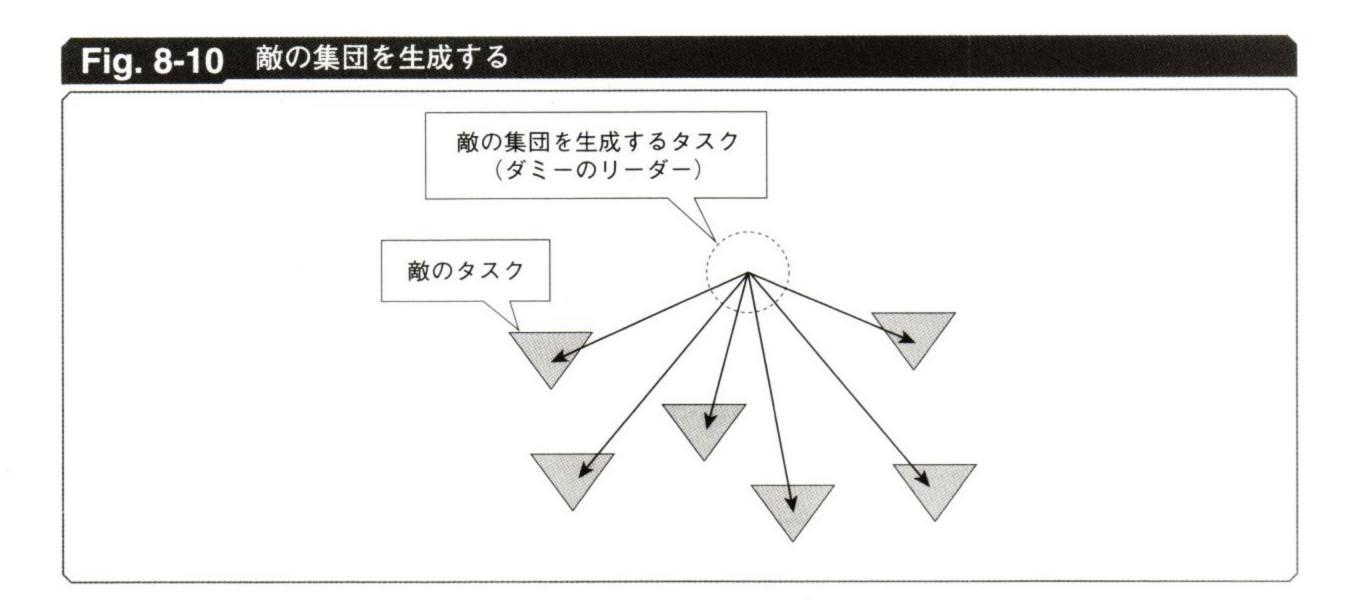
編隊の考え方を応用して、集団になった敵を出現させることもできます。これまでに紹介した編隊と異なるのは、リーダーがいない点です。

カッパ巻きの例では、リーダーが仲間となる小さな敵を生成していました。集団の場合には、いわばダミーのリーダーを生成し、このダミーのリーダーに集団を構成する敵を生成させます（Fig. 8-10）。ダミーのリーダーは敵とほぼ同じものですが、画面には表示せず、当たり判定もありません。

集団は一気に出現するのでもかまいませんし、一定時間の間に次々と敵が現れるパターンもあるでしょう。集団の生成が完了したら、ダミーのリーダーは消去します。

スクリプトに敵生成コマンドを並べれば、敵の集団を生成することはできます。敵の数

がある程度までならばこの方法でも十分ですが、数十体や数百体の敵からなる集団を出現させるときには、こういったダミーのリーダーを使うのが便利です。

**Fig. 8-10　敵の集団を生成する**

## ステージのスクリプト

今までのプログラムでは、敵を出現させるタイミングや種類を乱数で決めていました。このようにランダムに敵を出現させるゲームもありますが、多くのゲームでは決まったタイミングに決まった種類の敵が出現します。

どのタイミングでどの種類の敵を出すのかというデータのことを、「敵の出現シーケンス」と呼びます。一般にシューティングゲームでは、敵の出現シーケンスを作成する必要があります。

敵の出現シーケンスは、例えば次のようなものです。

赤身をX座標30に出現させる
20フレーム待つ
玉子をX座標-10に出現させる
100フレーム待つ

こういった敵の出現シーケンスは、プログラムで記述することもできますが、外部のデータファイルにすると便利です。例えば、以下のような内容のテキストファイルを用意しておき、これをプログラムから読み込んで使います。

```
enemy akami 30
wait 20
enemy tamago -10
wait 100
```

このデータファイルには敵の出現シーケンスだけではなく、時間待ち、BGMの再生や停止、メッセージの表示といった、ゲームの進行に関するさまざまなデータを書き込んでおくと便利です。このデータファイルが、ゲームの「スクリプト」に相当します。スクリプトとは台本や脚本という意味の言葉です。多くのゲームでは、スクリプトを使ってゲームの進行を記述します。

スクリプトを使うと、次のような利点があります。

### プログラマー以外のスタッフが編集しやすい

プログラムに直接ゲームの進行を記述してしまうと、プログラマー以外が編集することが難しくなります。しかしスクリプトを外部データファイルとして用意すれば、プログラマー以外のスタッフでも比較的容易に編集できます。作業をうまく分担すれば、プログラマーの負担を軽くすることができます。

### ゲームバランスの調整が簡単

ゲームバランスを調整するためには、敵の出現シーケンスを変更してはテストプレイする、という手順を何度も繰り返す必要があります。プログラムに出現シーケンスを記述すると、変更のたびにプログラムを再コンパイルしなくてはなりません。その点スクリプトならば、再コンパイルの手間が省けるので、作業がスムーズに進みます。

※

スクリプトを解釈して実行するためのプログラムを作る必要はありますが、作業の分担やゲームバランスの調整を考えると、全体としてはスクリプトを使った方が手間が減ります。特に、プログラマー以外のスタッフが敵の出現シーケンスを作ったりゲームバランスを取ったりする場合には、スクリプトを使うと効果的です。

一方、自分1人でゲームを制作する場合には、スクリプトを使うかどうかは、好みに応じて決めればよいと思います。スクリプトのためのルーチン作成に手間取ってゲームの開発自体が頓挫してしまっては残念ですので、必要だと感じたときに導入するのがお勧めです。

## スクリプトの例

本書のサンプルでは、スクリプトを使ってステージの進行を記述することにします。List 8-8は、スクリプトの記述例です。スクリプトは「script.txt」という名前のテキストファイルに記述します。

このスクリプトにはTable 8-1のようなコマンドがあります。ここでは、敵の生成に加えて、時間待ちやBGM再生に関するコマンドも用意しました。

Table 8-1　スクリプトのコマンド

| コマンド | 敵の生成（enemy） |
|---|---|
| 文法 | enemy 敵の名前 X座標 |
| 機能 | 指定した種類の敵を、指定したX座標に出現させる。Y座標は敵の種類に応じて自動的に設定 |
| コマンド | 時間待ち（wait） |
| 文法 | wait フレーム数 |
| 機能 | 指定されたフレーム数だけ、スクリプトの実行を中止する |
| コマンド | BGM再生（play） |
| 文法 | play BGM番号 |
| 機能 | 指定されたBGMを再生する |
| コマンド | BGMのフェードアウト（fadeout） |
| 文法 | fadeout 時間 |
| 機能 | BGMを指定された時間でフェードアウトさせる（音量を0までなめらかに下げる） |
| コマンド | ゲームの終了（gameover） |
| 文法 | gameover |
| 機能 | ゲームオーバー画面を表示して、ゲームを終了する |

**List 8-8　スクリプトの例（script.txt）**

```
//―――――――――――――――――――――――――――――――――――――――①
// BGMの0番を再生し、60フレーム待機する
play 0
wait 60

//―――――――――――――――――――――――――――――――――――――――②
// 赤身を4個出現させる
enemy akami 10
wait 10
enemy akami 20
wait 10
enemy akami 30
wait 10
enemy akami 40
wait 50

// ...（中略）...
```

```
//―――――――――――――――――――――――――――――――――――――――――――――――――③
// 大きな赤身と、玉子を8個出現させる
enemy bigakami -30
wait 50
enemy tamago 20
wait 10
enemy tamago 20
wait 10
enemy tamago 20
wait 10
enemy tamago 20
wait 10
enemy tamago 20
wait 10
enemy tamago 20
wait 10
enemy tamago 20
wait 10
enemy tamago 20
wait 150

// ...（中略）...

//―――――――――――――――――――――――――――――――――――――――――――――――――④
// BGMをフェードアウトし、180フレーム待機する
fadeout 180
wait 180

//―――――――――――――――――――――――――――――――――――――――――――――――――⑤
// ゲームを終了する
gameover
```

## スクリプトの要点

List 8-8のスクリプトについて要点を解説します。

### ① BGMの再生

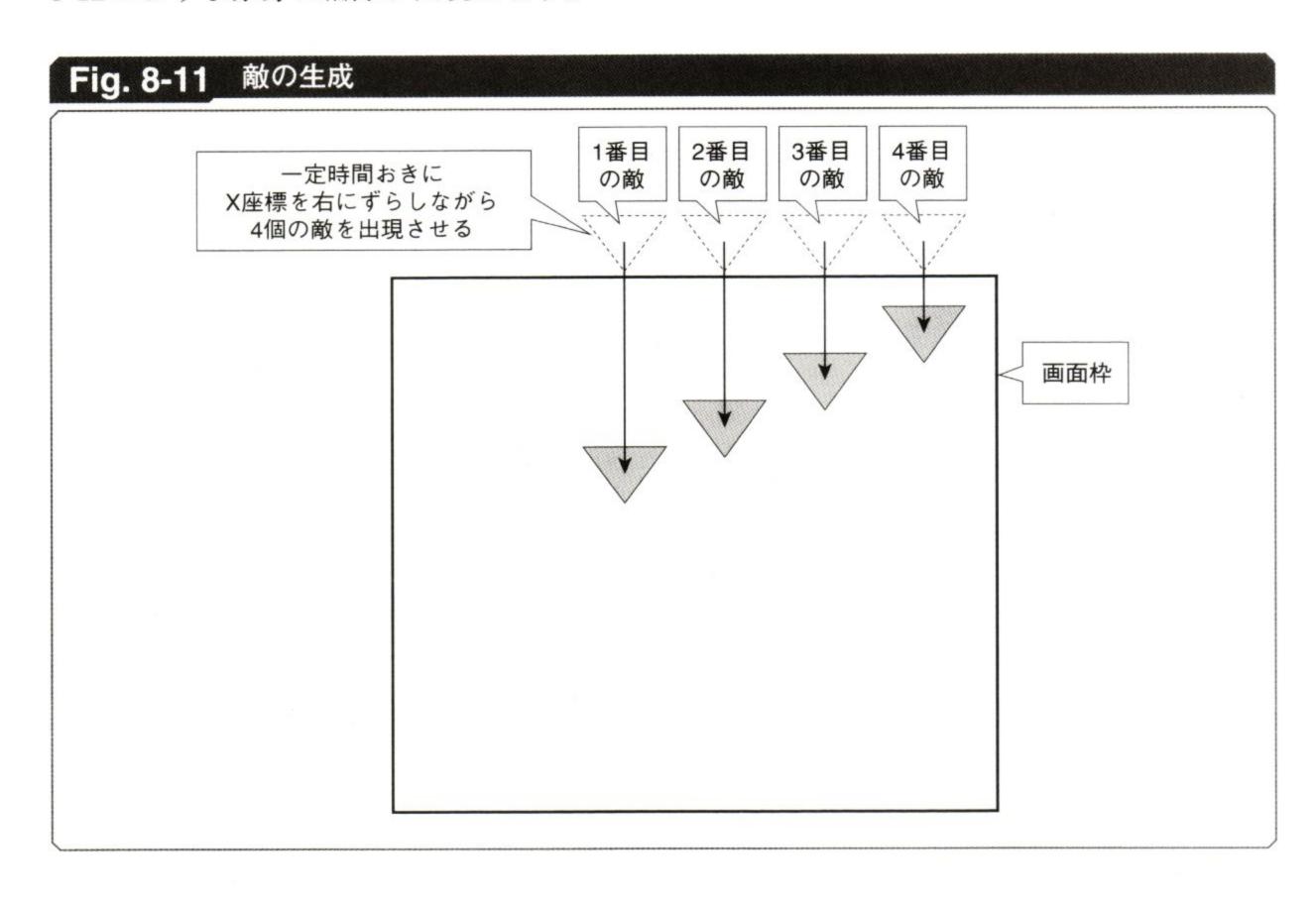
playコマンドを使って0番のBGMを再生します。また、waitコマンドで60フレーム待機します。

### ② 敵の生成

赤身（akami）を4個出現させます。出現時のX座標は10から40まで変化させます。敵を1

個生成するたびに、waitコマンドで10フレーム待機します(Fig. 8-11)。結果として、Fig. 8-12のような赤身の編隊が出現します。

Fig. 8-11　敵の生成

Fig. 8-12　生成された敵の編隊

### ③ 複数種類の敵を組み合わせて生成

②とは別の敵を出現させます。ここでは大きな赤身（bigakami）と、玉子（tamago）を生成します。結果はFig. 8-13です。このように、複数種類の敵を組み合わせて出現させることもできます。

Fig. 8-13 複数種類の敵の組み合わせ

### ④ BGMのフェードアウト

ステージの終わりで、fadeoutコマンドを使ってBGMをフェードアウトさせます。そしてフェードアウトの時間待ちをします。

### ⑤ ゲーム終了

gameoverコマンドを使ってゲームオーバー画面を表示し、ゲームを終了します。次のChapter 9では、ゲームオーバーにするかわりにここでボスを出現させる方法を解説します（P. 284）。

## スクリプト実行用のクラス構成

スクリプトを実行するために、Fig. 8-14のようなクラス群を作りました。各クラスの役割は以下のとおりです。

### CCommand

スクリプトに記述するコマンドの抽象基底クラスです。コマンドを実行する純粋仮想関数Runを定義します。各コマンドのクラスはCCommandクラスから派生します。

### CEnemyCommand

敵を生成するコマンドです。CCommandクラスから派生し、Run関数をオーバーライドします。

### CWaitCommand

時間待ちをするコマンドです。CCommandクラスから派生し、Run関数をオーバーライドします。

### CPlayCommand

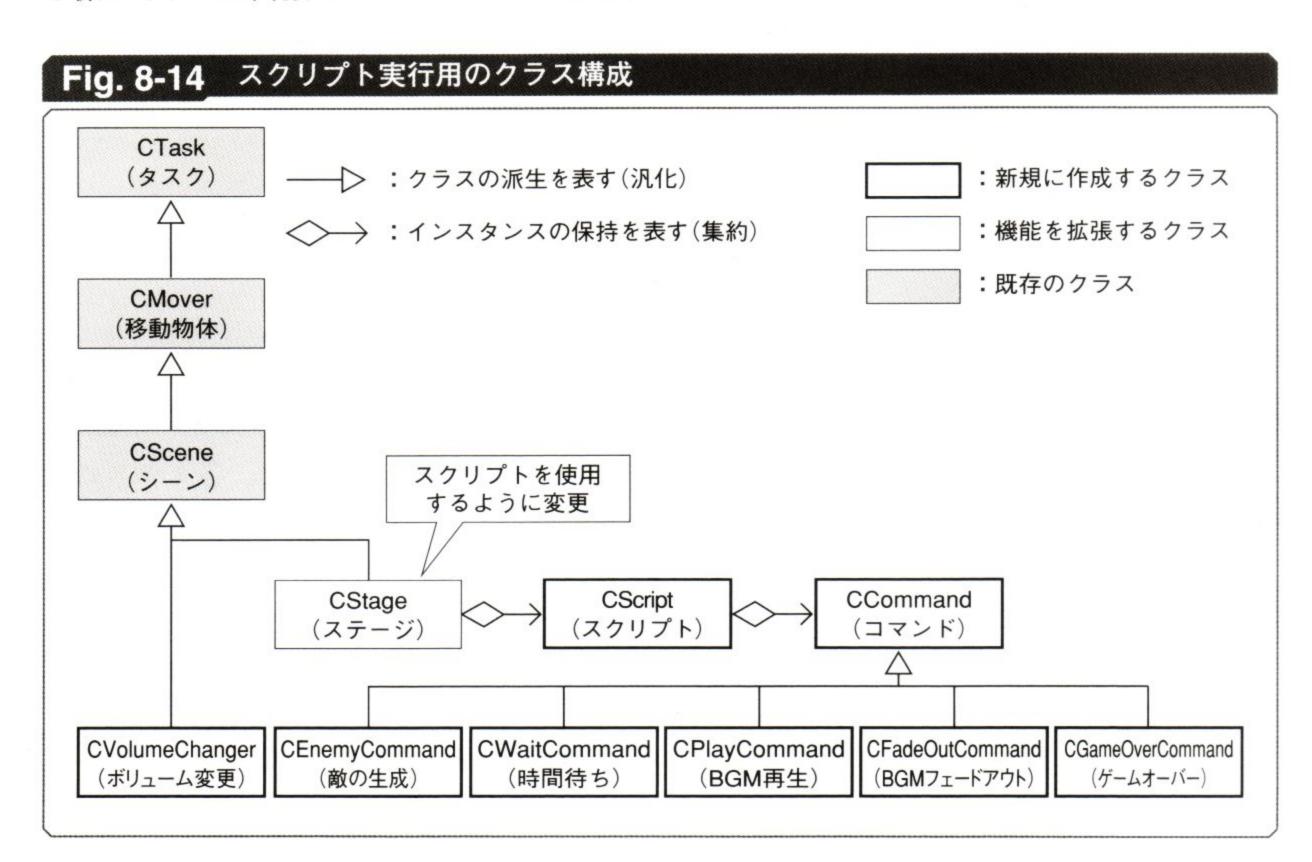
BGMを再生するコマンドです。CCommandクラスから派生し、Run関数をオーバーライドします。

### CFadeOutCommand

BGMをフェードアウトさせるコマンドです。CCommandクラスから派生し、Run関数をオーバーライドします。

### CGameOverCommand

ゲームオーバー画面を表示して、ゲームを終了するコマンドです。CCommandクラスから派生し、Run関数をオーバーライドします。

Fig. 8-14 スクリプト実行用のクラス構成

### CScript

スクリプトです。コマンド列（CCommandから派生したクラスのインスタンスの配列）を保持し、コマンドを順番に実行します。

### CStage

Chapter 7で作成したステージのクラスです（P. 221）。CScriptクラスを使用してステージを進行するように変更します。

### CVolumeChanger

指定された時間をかけてBGMのボリュームを変更するクラスです。BGMのフェードアウトに使います。CSceneクラスから派生します。

## スクリプトの仕組み

スクリプトは、Fig. 8-15のような方法で実行しています。スクリプトを解釈して配列に変換する過程と、配列からコマンドを取り出して実行する過程があります。

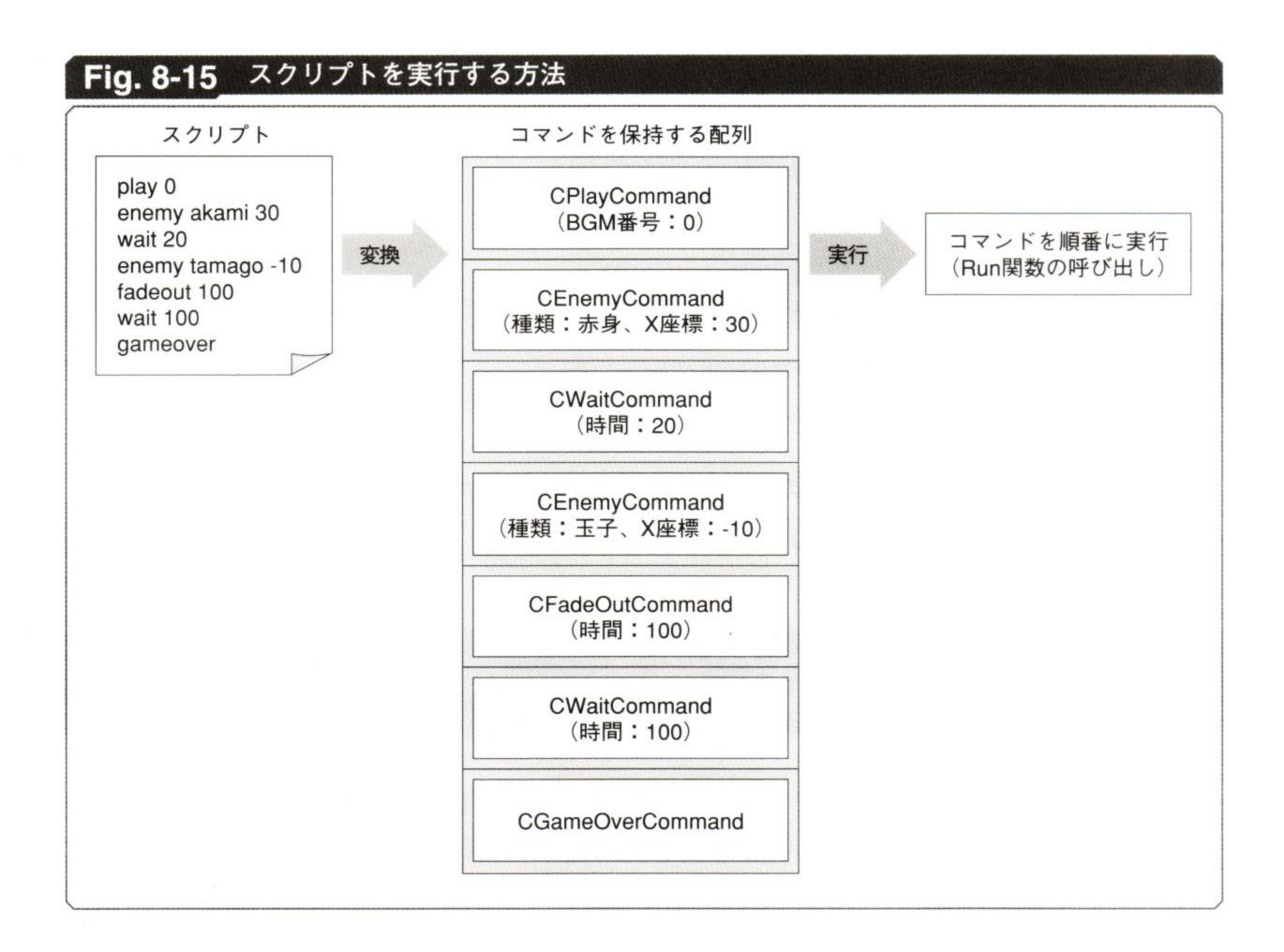
まず、テキストファイルとして用意されたスクリプトを読み込み、CCommandから派生したクラスのインスタンスの配列に変換します。例えば、playコマンドならばCPlayCommandインスタンス、enemyコマンドならばCEnemyCommandインスタンスを生成して、配列に格納します。これらはCCommandクラスの派生クラスなので、CCommandクラスの配列に格納することができます。

スクリプトを実行するには、配列からコマンドを順に取り出して、Run関数を呼び出します。Run関数は仮想関数で、各コマンドのクラスではRun関数をオーバーライドしています。そのため、Run関数を実行するとコマンドに応じた処理が実行されます。

CCommandクラスのような基底クラスを用意し、そこから各コマンドのクラスを派生させ、Run関数のようなコマンド実行用の関数をオーバーライドすることが、この方法のポイントです。なお、これはデザインパターンのCommandパターンに相当します。

このように実装すると、コマンドの追加が容易になります。新しいコマンドのクラスを追加しても、コマンドを実行する仕組みを変更する必要がないからです。

ゲームの開発が進むにつれて、きっとスクリプトに新しいコマンドを増やす必要が出てくるでしょう。コマンドを追加しやすい仕組みにしておけば、そういった要求にも素早く対応することができます。

**Fig. 8-15 スクリプトを実行する方法**

## スクリプトを配列に登録する

ファイルからロードしたスクリプトをコマンドの配列に変換するには、コマンドを解釈する処理が必要です。それにはまず、スクリプトのコマンド行からコマンド名（enemyなど）とパラメータ（akamiや30など）を切り出します。そして、コマンド名に応じたコマンド（CCommandクラスの派生クラスのインスタンス）を作成して、配列に追加します。

また、スクリプトにはコメントを書けるようにしました。/*と*/で囲った範囲はコメントになり、解釈時には無視されます。コメントは、例えば以下のように書きます。

```
/*
enemy akami 10
wait 10
enemy akami 20
wait 10
*/
```

コメントは注釈を書くためというよりも、むしろテストプレイを迅速に行うために役立ちます。ステージがある程度長くなると、通してプレイするのに時間がかかるため、ステージの最後の方の敵出現シーケンスを調整するのは大変です。そこでコメントを使って、調整したい部分以外をコメントアウトすることにより、調整したい部分をピンポイントでテストすることができます。

List 8-9は、スクリプトを解釈するプログラムです。コマンドを保持する配列には、可変長配列（標準C++ライブラリのvector）を使いました。

ゲームで可変長配列を使うときには注意が必要です。可変長配列の使い方によっては、使用するメモリの総量が実行時までわからないプログラムになることがあります。つまり、ある環境で動くか動かないのかが、いざ実行してみるまでわからないゲームになってしまう可能性があります。特に、タスクの管理などに可変長配列を使うのは要注意です。

しかし、スクリプトの内容は実行前、さらにいえばゲームの公開前に決定しておくケースがほとんどでしょう。その場合には、使用するメモリの総量をあらかじめ見積もることができるので、簡単にするためにここではvectorを使用することにしました。

**List 8-9　スクリプトの解釈（Script.h、Script.cpp）**

```
// スクリプト
class CScript {

    // コマンドを保持する配列
    vector<CCommand*> Command;

    // 実行中のコマンド番号、待機時間
    int CommandIndex, Wait;

public:

    // コンストラクタ
    CScript(string file);

    // 実行開始、実行、待機時間の設定
    void Init();
    void Run();
    void SetWait(int wait) { Wait=wait; }
};

// enemyコマンドで指定することができる敵の名前の一覧
// CEnemyCommandインスタンスを生成するときに使用
const static char* ENEMY_NAME[]={
    "akami", "tamago", "ebi",
```

```
    "bigakami", "bigtamago", "bigebi",
    "bigtekka", "bigkappa", "bigkampyo"
};

// 敵を生成する関数への関数ポインタのテーブル
// CEnemyCommandインスタンスを生成するときに使用
const static NEW_ENEMY_FUNC ENEMY_FUNC[]={
    CAkami::New, CTamago::New, CEbi::New,
    CBigAkami::New, CBigTamago::New, CBigEbi::New,
    CBigTekka::New, CBigKappa::New, CBigKampyo::New
};

// コンストラクタ
CScript::CScript(string file)
:   CommandIndex(0), Wait(0)
{
    // スクリプトをファイルからロード
    // CStringsクラスで複数行の文字列を扱う
    // CStringsクラスの詳細は付録CD-ROMのLibUtil¥Util.hとLibUtil¥Util.cppに記述
    CStrings ss;
    ss.LoadFromFile(file);

    // スクリプトを1行ずつ解釈する
    bool comment=false;
    for (int i=0, in=ss.GetCount(); i<in; i++) {

        // コマンドとパラメータの取り出し
        string
            s=ss.GetString(i),
            command=GetToken(s, 0, " "),
            param0=GetToken(s, 1, " "),
            param1=GetToken(s, 2, " ");

        // コメントの処理
        if (command=="*/") comment=false; else
        if (command=="/*") comment=true;
        if (comment) continue;

        // 敵の生成
        if (command=="enemy") {
            for (int j=0, jn=sizeof(ENEMY_NAME)/sizeof(char*);
                j<jn; j++
            ) {
                if (param0==ENEMY_NAME[j]) {

                    // コマンドを作成して配列の末尾に追加
```

```
                Command.push_back(
                    new CEnemyCommand(
                        ENEMY_FUNC[j], ToFloat(param1)));
                break;
            }
        }
    } else

    // 時間待ち
    if (command=="wait") {
        Command.push_back(
            new CWaitCommand(this, ToInt(param0)));
    } else

    // BGM再生
    if (command=="play") {
        Command.push_back(
            new CPlayCommand(ToInt(param0)));
    } else

    // BGMフェードアウト
    if (command=="fadeout") {
        Command.push_back(
            new CFadeOutCommand(ToInt(param0)));
    }

    // ゲームオーバー
    if (command=="gameover") {
        Command.push_back(
            new CGameOverCommand());
    }
  }
}
```

## 敵を生成する関数

敵を生成する際には、その敵クラスのコンストラクタを呼び出したいところです。敵の種類が多いので、処理を簡潔にするには関数ポインタを使うのがよい方法なのですが、残念ながらコンストラクタについては関数ポインタが使えません。例えば、赤身のクラスのコンストラクタを関数ポインタを用いて呼び出すことはできません。

そこで、敵を生成するための関数を敵のクラスに用意しておきます。これはコンストラクタを呼び出すだけの関数です。そして、コンストラクタのかわりに、この関数への関数ポインタを利用します。

なお、敵を生成する関数は、静的メンバ関数にすることがポイントです。敵のインスタンスを生成する前なので、静的ではないメンバ関数は呼び出すことができません。

敵を出現させるコマンド（CEnemyCommandクラスのインスタンス）は、敵の種類に応じた関数ポインタを保持しています。この関数ポインタを使って敵の生成関数を呼び出せば、コマンドに指定された種類の敵が出現するというわけです。先のList 8-9では、敵を生成する関数への関数ポインタのテーブル（配列ENEMY_FUNC）を使って、コマンドに関数ポインタを登録しています。

List 8-10は、敵を生成する関数の例です。これは赤身の例ですが、同様にすべての敵について生成用の関数を用意します。

**List 8-10　敵を生成する関数（Enemy.h、Script.h）**

```
// 赤身
class CAkami : public CSmallEnemy {
public:
    CAkami(float x);
    virtual bool Move();

    // 敵を生成する関数
    static CEnemy* New(float x) { return new CAkami(x); }
};

// 敵を生成する関数へのポインタ
// 関数ポインタは複雑な型になりがちなので、typedefを使うと便利
typedef CEnemy* (*NEW_ENEMY_FUNC)(float x);
```

## スクリプトの実行

スクリプトを実行するには、コマンドを登録した配列からコマンドを1つずつ取り出して実行します。各コマンドのRun関数を呼び出すと、そのコマンドが実行されます。

待機時間に関する処理もここで行います。待機するよう指示された時間が経過していない間は、コマンドを実行せずに、タイマーだけを更新します。

List 8-11は、スクリプトを実行するプログラムです。

**List 8-11　スクリプトの実行（Script.cpp）**

```
// 初期化
void CScript::Init() {

    // コマンド番号と待機時間を初期化
```

```
    CommandIndex=0;
    Wait=0;
}

// 実行
void CScript::Run() {

    // 待機時間を更新
    if (Wait>0) Wait--;

    // 待機中でなければコマンドを実行
    while (CommandIndex<(int)Command.size() && Wait==0) {

        // コマンドを実行する
        Command[CommandIndex]->Run();

        // コマンド番号を更新して、次のコマンドへ進む
        CommandIndex++;
    }
}
```

## スクリプトを用いたステージの進行

ゲームのステージを表すクラス（CStage）を、スクリプトに対応させておきましょう（List 8-12）。以前は乱数で敵を生成していましたが、フレームごとにスクリプトの実行処理（CScript::Run関数）を呼び出すように変更します。

このように、スクリプトを使って敵の生成やBGMの再生などを行うと、ステージの進行処理をとてもシンプルにすることができます。細かい処理はすべてスクリプトのなかで制御するため、ステージの進行処理では単にスクリプトを実行するだけですみます。

**List 8-12　スクリプトを用いたステージの進行（Scene.cpp）**

```
// コンストラクタ
CStage::CStage()
:   PrevInput(true)
{
    // スクリプトの最初から実行するため、
    // CScript::Init関数（List 8-11）を呼び出す
    Game->Script->Init();
}
```

```
// 移動
bool CStage::Move() {

    // ポーズ
    const CInputState* is=Game->GetInput()->GetState(0);
    if (!Game->IsPaused() && !PrevInput && is->Button[2]) {
        new CPause();
    }
    PrevInput=is->Button[2];
    if (Game->IsPaused()) return true;

    // スクリプトの実行（List 8-11）
    Game->Script->Run();

    return true;
}
```

# スクリプトのコマンド

ここからは各種のコマンドと、その仕組みについて簡単に解説します。

## 敵生成コマンド

敵生成コマンドは、敵を生成する関数へのポインタと、敵のX座標を保持する必要があります。敵を生成する際には、この関数ポインタを用いて敵の生成処理を呼び出します。

List 8-13が、敵生成コマンドを表すクラス（CEnemyCommand）です。

**List 8-13 敵生成コマンド（Script.h）**

```
class CEnemyCommand : public CCommand {

    // 敵を生成する関数、敵のX座標
    NEW_ENEMY_FUNC Func;
    float X;

public:

    // コンストラクタ
    CEnemyCommand(NEW_ENEMY_FUNC func, float x)
    :   Func(func), X(x) {}
```

```
    // コマンドの実行
    virtual void Run() { Func(X); }
};
```

## 時間待ちコマンド

時間待ちコマンドは、指定された待機時間を設定します。待機時間を設定する機能はスクリプトのクラス（CScript）にあります。待機時間が設定されていると、スクリプトのクラスはスクリプトを進行せずに、待機時間の更新だけを行います。

List 8-14は、時間待ちコマンドを表すクラス（CWaitCommand）です。

**List 8-14　時間待ちコマンド（Script.h）**

```
// 時間待ちコマンド
class CWaitCommand : public CCommand {

    // スクリプト、待ち時間
    CScript* Script;
    int Wait;

public:

    // コンストラクタ、コマンドの実行
    CWaitCommand(CScript* script, int wait)
    :   Script(script), Wait(wait) {}
    virtual void Run() { Script->SetWait(Wait); }
};

// 待機時間の設定
void CScript::SetWait(int wait) {
    Wait=wait;
}
```

## BGMの再生

BGMの再生を行うため、次のような機能を実装しました。BGMのコントロールには、Chapter 2で作成したメディアクラス（P. 43）を使います。

### 再生

BGMをロードし、ボリュームを設定してから再生します。再生するBGMは番号で指定

することにしました。プログラムには番号とファイル名の対応を表すテーブルを用意しておきます。

### 停止

BGMを停止します。

### 一時停止と再開

BGMを一時停止したり、一時停止しているBGMを再開したりします。

### ボリュームの設定

BGMのボリュームを設定します。BGMのフェードアウトなどに利用します。

※

List 8-15は、BGMの再生に関するプログラムです。

**List 8-15　BGMの再生（Main.cpp）**

```
// BGMのファイル名
const static char* bgm_file[]={
    "stage1", "stage2", "stage3", "stage4",
    "boss1", "boss2", "ending1", "ending2"
};

// 再生
void CShtGame::PlayBGM(int i) {
    BGMPlayer->LoadFromFile(
        ExePath+"..¥¥Music¥¥"+bgm_file[i]+".mid");
    BGMPlayer->SetVolume(1);
    BGMPlayer->Play();
}

// 停止
void CShtGame::StopBGM() {
    BGMPlayer->Stop();
}

// 一時停止と再開
void CShtGame::PauseBGM(bool pause) {
    if (pause) BGMPlayer->Pause(); else BGMPlayer->Play();
}

// ボリュームの設定
void CShtGame::SetBGMVolume(float v) {
    BGMPlayer->SetVolume(v);
}
```

## BGM再生コマンド

List 8-15で作成したBGM再生機能を利用して、BGM再生コマンドを実現しました。このコマンドは再生するBGM番号を保持しています。

List 8-16は、BGM再生コマンドを表すクラス（CPlayCommand）です。

**List 8-16 BGM再生コマンド（Script.h）**

```
class CPlayCommand : public CCommand {

    // BGM番号
    int BGMIndex;

public:

    // コンストラクタ、コマンドの実行
    CPlayCommand(int bgm_index) : BGMIndex(bgm_index) {}
    virtual void Run() { Game->PlayBGM(BGMIndex); }
};
```

## BGMフェードアウトコマンド

BGMフェードアウトコマンドは、フェードアウト時間を保持します。そして、後述するボリューム変更クラス（List 8-18）を利用して、時間とともにボリュームを少しずつ下げることにより、フェードアウトを行います。

フェードアウト時間はフレーム単位で指定します。例えば60を指定すると、60フレーム（約1秒）かけてBGMのボリュームを1（最大）から0（最小）まで変化させます。

List 8-17は、BGMフェードアウトコマンドを表すクラス（CFadeOutCommand）です。

**List 8-17 BGMフェードアウトコマンド（Script.h）**

```
class CFadeOutCommand : public CCommand {

    // フェードアウト時間
    int Time;

public:

    // コンストラクタ、コマンドの実行
    CFadeOutCommand(int time) : Time(time) {}
    virtual void Run() { new CVolumeChanger(Time, 1, 0); }
};
```

## BGMのフェードアウトとフェードイン

BGMのフェードアウトやフェードインを実現するには、一定の時間をかけてBGMのボリュームをしだいに変化させる必要があります。こういった処理には、タスクシステムを利用すると便利です。ここではタスクシステムを用いて、フレームごと（1/60秒などの一定時間ごと）にボリュームを設定することにより、BGMのボリュームをしだいに変化させます。

最初に、目標のボリュームと時間から、ボリュームの変化を計算しておきます。そしてフレームごとにBGMのボリュームを設定し、ボリュームと時間を更新します。一定時間が経過したら、タスクを消去することによって、ボリュームの変更を終了します。

List 8-18は、BGMのボリュームを変化させるクラス（CVolumeChanger）です。

**List 8-18　BGMボリュームを変化させるクラス（Scene.h、Scene.cpp）**

```
// BGMボリュームを変化させるクラス
// 画面の基本クラス（CScene）から派生
class CVolumeChanger : public CScene {
protected:

    // 時間、現在のボリューム、ボリュームの変化
    int Time;
    float Volume, VVolume;

public:

    // コンストラクタ、移動
    CVolumeChanger(int time, float from, float to);
    virtual bool Move();
};

// コンストラクタ
CVolumeChanger::CVolumeChanger(int time, float from, float to)
:   Time(time), Volume(from)
{
    // ボリュームの変化を計算
    VVolume=to-from;
    if (time>1) VVolume/=(time-1);
}

// 移動
bool CVolumeChanger::Move() {

    // ボリュームの設定
```

```
    Game->SetBGMVolume(Volume);

    // ボリュームと時間の更新
    Volume+=VVolume;
    Time--;

    // 指定時間が経過したら終了（タスクの消去）
    // タスクの消去はCShtGame::MoveTask関数で行う（P. 112）
    return Time>0;
}
```

## ゲーム終了コマンド

ゲーム終了コマンドは、ゲームオーバー画面（P. 232）を生成します。

List 8-19は、ゲーム終了コマンドを表すクラス（CGameOverCommand）です。

**List 8-19　ゲーム終了コマンド（Script.h）**

```
class CGameOverCommand : public CCommand {
public:

    // コンストラクタ、コマンドの実行
    CGameOverCommand() {}
    virtual void Run() { new CGameOver(); }
};
```

## ゲーム停止時と終了時のBGM操作

ポーズやコンティニューのためにゲームが一時停止したときには、BGMを一時停止します。ゲームを再開したときには、BGMも再開します。また、ゲームオーバーになったときには、BGMも停止します。

List 8-20は、ゲーム停止時と終了時のBGM操作に関するプログラムです。BGMの操作にはList 8-15の関数群を使用します（P. 272）。

**List 8-20　BGMの操作（Main.h、Main.cpp）**

```
// ゲームの一時停止
// BGMの一時停止と再開に関する処理を追加
void CShtGame::SetPaused(bool paused) {
    Paused=paused;
    PauseBGM(paused);
```

```
}

// ゲーム本体の動作
// BGMの停止に関する処理を追加
void CShtGame::Move() {

    // タスクの動作
    // ...（中略）...

    // ゲームオーバー処理
    if (GameEnd) {

        // BGMの停止
        StopBGM();

        // タスクの消去
        BackList->DeleteTask();
        BeamList->DeleteTask();
        BulletList->DeleteTask();
        EffectList->DeleteTask();
        EnemyList->DeleteTask();
        MyShipList->DeleteTask();
        ShotList->DeleteTask();
        SceneList->DeleteTask();

        // タイトル画面へ
        new CTitle();
        GameEnd=false;
    }
}
```

## >>Chapter 8のまとめ

本章ではステージを彩る背景と敵のパターンを作成し、ステージの進行を記述・管理するスクリプト機能を実装しました。あなたが作ろうと思っているシューティングゲームは、どんなゲームでしょうか。ゲームによっては、ここまでの解説でもう十分作れるようになっているかもしれません。次章から解説するボスやアイテムなどは、ゲームのバランスや方針に応じて、適宜取り入れていただければよいと思います。

# Chapter 09

# ボス

最近のほとんどのシューティングゲームでは、各ステージの最後にボスと呼ばれる巨大な敵が出現します。ボスは、高い耐久力と強力な攻撃力を備えています。
本章のテーマは、ステージの最後を盛り上げるボスです。次のステージに進むためには、強大なボスを倒さなくてはならないため、ボスを上手に使うとゲームの緊張感を高めることができます。

# ボス

昨今では、ほとんどのシューティングゲームにボスや中ボスと呼ばれる敵が出現します。ザコと呼ばれる通常の敵とは違い、ボスは耐久力が非常に高く、激しい攻撃をしてきます。また、非常に大型のボスや、合体や変形をするボスなども珍しくありません。

多くのゲームではステージの最後にボスが出現し、ボスを倒すとステージクリアとなります。中ボスはステージの中間付近で出現することが多いようです。

ボスや中ボスが出現することによって、ステージの展開にメリハリがつきます。また、強力なボスを倒さないと次のステージに進めない、ということがプレイヤーの意欲をかき立てます。さらに、ボスの形状・動作・弾幕はそれ自体が面白いことが多いため、先のステージに進んで新しいボスを見ることがプレイヤーにとっても楽しみになります。

本章ではボスを制作します（Fig. 9-1）。見た目はあまり強そうではありませんが、弾幕はボスらしく激しいものにしました。外見とは裏腹になかなか手強いはずです。

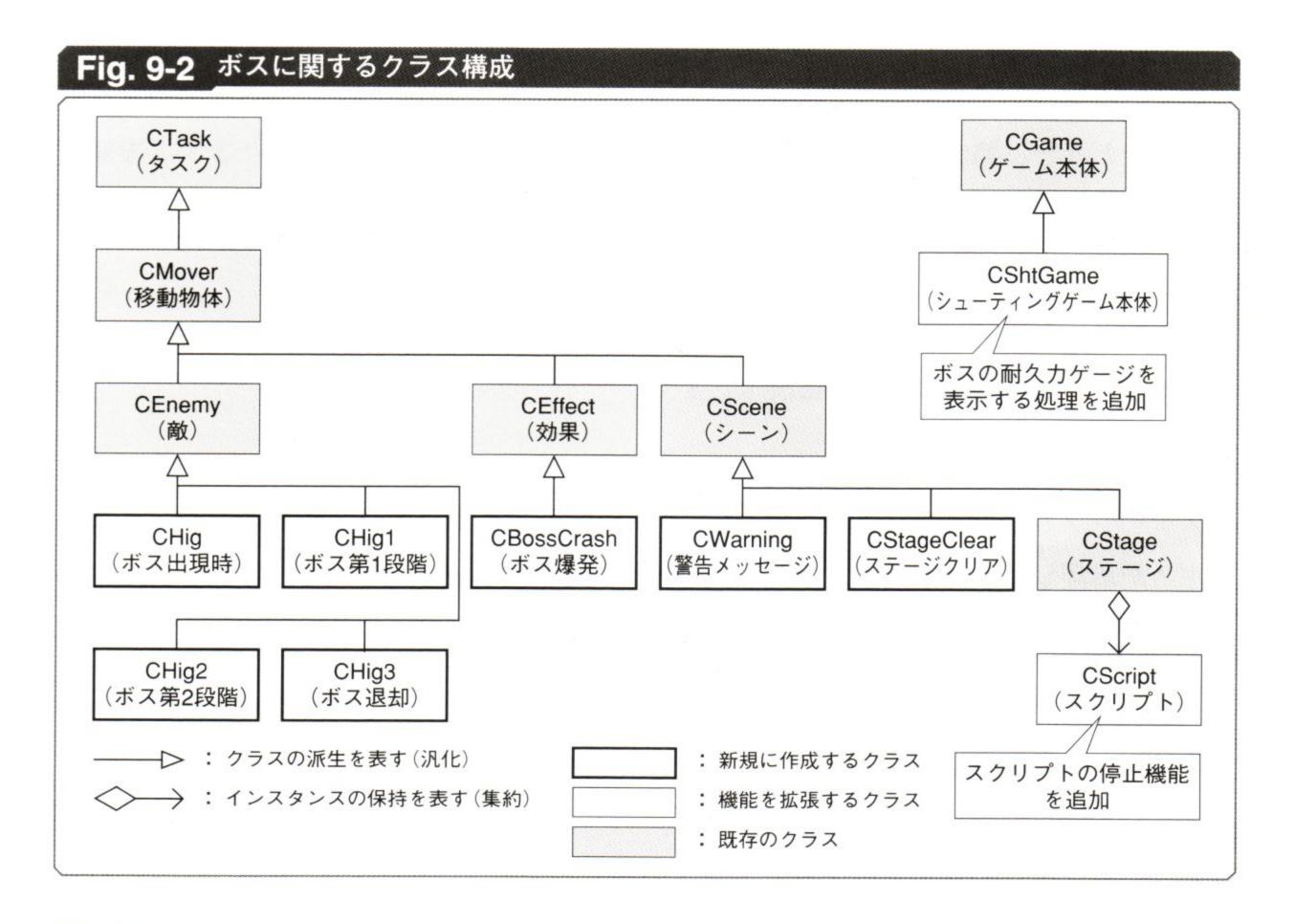
本章では、ボスの出現・移動・攻撃といった処理の作り方を解説します。また、ボスの出現前に警告メッセージを出したり、ボスの耐久力をグラフ表示したり、ダメージを受けたボスの攻撃パターンを変化させたりといった処理についても説明します。

本章のプロジェクトは「ShtGame_Boss」フォルダに収録してあります。実行ファイルは「ShtGame_Boss¥Release¥ShtGame.exe」です。

**Fig. 9-1** ボス

## ボスに関するクラス

Fig. 9-2はボスに関するクラス構成です。ボスの各段階に対応したクラスの他、ボスの爆発や警告メッセージなどのクラスがあります。各クラスの役割は次のとおりです。

**Fig. 9-2 ボスに関するクラス構成**

### CHig

出現時のボスです。敵を表すCEnemyクラス(P. 189)から派生します。ボスは画面上方から出現します。このときは無敵状態です。なお、クラス名については、ボスのデザインがヒグマなのでCHigとしました。

### CHig1

第1段階のボスです。CEnemyクラスから派生します。放射状に弾を発射して攻撃します。

### CHig2

第2段階のボスです。CEnemyクラスから派生します。2体に分裂したボスが、放射状に弾を発射して攻撃します。

### CHig3

退却時のボスです。CEnemyクラスから派生します。時間内にボスを破壊できなかった場合、ボスは退却します。このときのボスは無敵状態です。

### CBossCrash

ボスの爆発です。効果を表すCEffectクラス（P. 165）から派生します。第1段階のボスを破壊したときと、第2段階のボスを破壊したときに、それぞれ爆発が起こります。爆発時には画面上の弾を消す処理も行います。

### CWarning

警告メッセージです。各種の画面を表すCSceneクラス（P. 214）から派生します。点滅するメッセージをボスの出現時に表示します。

### CStageClear

ステージクリア画面です。CSceneクラスから派生します。ボスを破壊するか、あるいはボスが退却したときに、ステージ完了のメッセージを表示します。

### CShtGame

シューティングゲームの本体となるクラスです（P. 111）。本章ではボスの耐久力ゲージを表示する処理を追加します。

### CScript

ステージを進行させるスクリプトのクラスです（P. 265）。本章ではスクリプトの一時停止と再開を行う機能を追加します。これはボスの出現時に通常の敵の出現を一時的に止めるために使います。

## ボスに関するタスクの生成関係

Fig. 9-3はボスに関するタスク（インスタンス）の生成関係を表しています。各タスクは以下のように生成されます。

### ボスの出現

スクリプトでボス出現コマンドが実行されると、ステージ（CStage）が出現時のボス（CHig）を生成します。この時点では、ボスはまだ画面に登場しません。

### 警告メッセージの表示

一定時間が経過すると、出現時のボス（CHig）が警告メッセージ（CWarning）を生成します。出現時のボスは画面外にいるので、画面には先に警告メッセージが表示されます。少し時間が経過すると、ボスも画面内に移動してきます。この段階では、まだボスに攻撃を当てることはできません。

### 第1段階のボス

一定時間が経過すると、出現時のボス（CHig）が第1段階のボス（CHig1）を生成します。ここでボスは攻撃を開始します。また、ボスに自機のショットやビーム、ボムが当たるようになります。

### 第2段階のボス

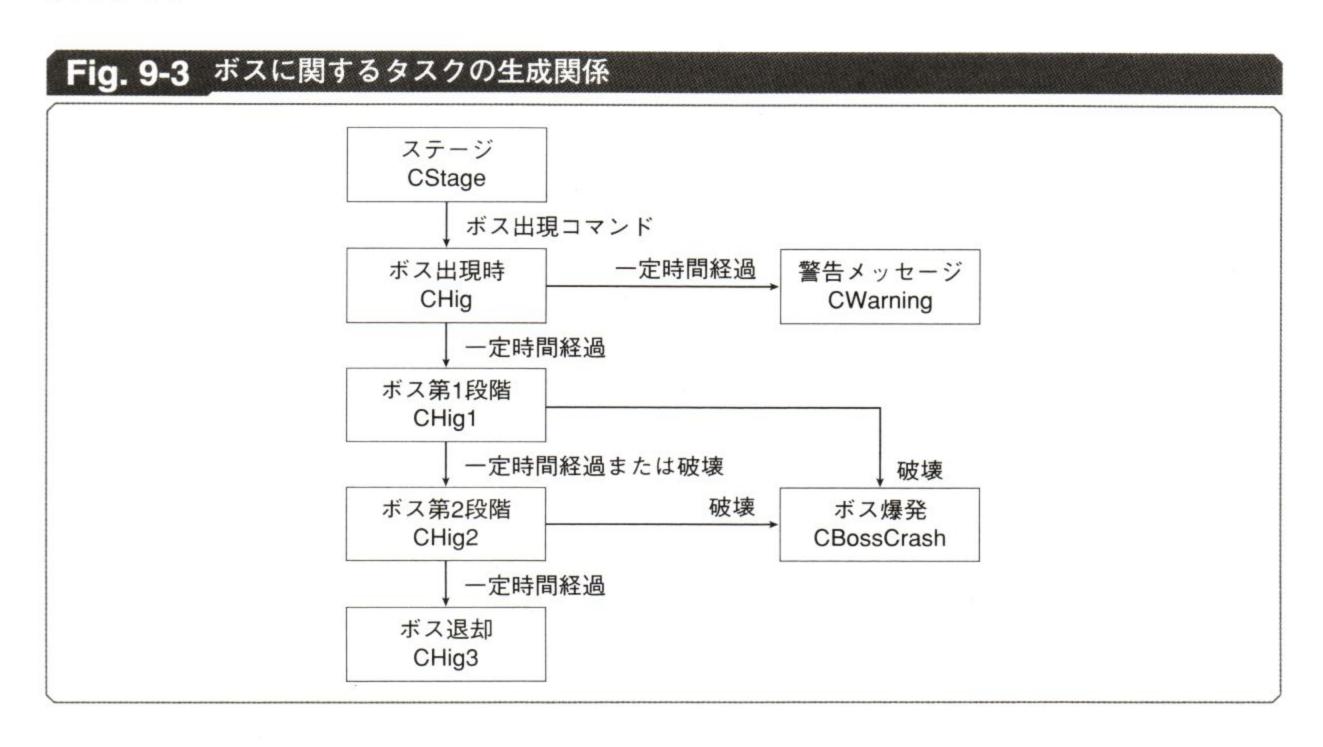
第1段階のボスが破壊されるか、破壊されなくても一定時間が経過すると、第2段階のボス（CHig2）を生成します。入れ替わりに、第1段階のボスは消去します。

### ボスの爆発

第1段階および第2段階のボスが破壊されたときには、ボスの爆発（CBossCrash）を生成します。

### ボスの退却

第2段階のボスが破壊されないまま一定時間が経過すると、退却時のボス（CHig3）を生成します。

**Fig. 9-3** ボスに関するタスクの生成関係

# 警告メッセージ

ボスに関する演出はゲームによってさまざまです。一般的には、ステージの最後に到達すると通常の敵が出現しなくなります。そして、これからボスが出現することをプレイヤーに警告するために、あるいは緊張感を高めたり雰囲気を盛り上げたりするために、警告メッセージを表示します。ゲームによっては次のような演出が入る場合もあります。

・ボスのセリフ
・主人公とボスとの会話
・ボスのイラストやムービーの表示

この他、建物が崩れたり地面が割れたり、あるいはスクロールが停止したりといった演出もあります。こういった演出を見るのはプレイヤーにとっても楽しみになるでしょう。ただし、あまり長々とした演出にすると何度もプレイしているうちにあきてしまうので、テンポのよいものにすることが肝要です。

## 警告メッセージを表示するプログラム

今回は、Fig. 9-4のような警告メッセージを表示することにしました。ボスが出現する直前に、メッセージを点滅させながら表示します。

ここでは、約1/3秒周期でメッセージを点滅させながら表示します。そして120フレーム、つまり約2秒間が経過したら、自分（警告メッセージ）のタスクを消去します。

**Fig. 9-4** 警告メッセージ

List 9-1は、警告メッセージを表示する処理をまとめたクラス(CWarning)です。少し手を加えれば、ボスごとに違ったメッセージを表示したり、メッセージの表示エフェクトを変えたりすることもできます。また、メッセージのかわりに画像を表示することも可能です。

**List 9-1 警告メッセージの表示(Scene.h、Scene.cpp)**

```
// 警告メッセージのクラス
// 各種の画面を表すクラス(CScene)から派生
class CWarning : public CScene {
protected:

    // タイマー
    int Time;

public:

    // コンストラクタ、移動、描画
    CWarning();
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CWarning::CWarning()
:   Time(0)
{}

// 移動
bool CWarning::Move() {

    // 一定時間が経過したら消去
    // Move関数でfalseを返すと
    // 呼び出し元のMoveTask関数がタスクを消去する
    Time++;
    return Time<120;
}

// 描画
void CWarning::Draw() {

    // 座標と色の設定
    int h=Game->GetGraphics()->GetHeight();
    D3DCOLOR w=ColWhite, s=ColShade;

    // メッセージを点滅表示させる
```

```
    // 約1/6秒ごとにメッセージの表示と非表示を切り替えるので、
    // メッセージが点滅して見える
    if (Time/10%2) {
        Game->Font->DrawText("WARNING",
            h/2-7*16/2, h/2-32, w, s);
        Game->Font->DrawText("A HUGE ENEMY IS APPROACHING",
            h/2-27*16/2, h/2, w, s);
    }
}
```

## ボスの出現

警告メッセージに続いて、実際にボスが出現します。ボスが画面端から出現するゲームもあれば、アルファブレンディングなどを使用して、画面内に浮き上がるようにボスを出現させるゲームもあります。ここでは、画面上端からボスを出現させることにしました(Fig. 9-5)。

画面上部で、ボスの耐久力をグラフ状のゲージで表しています。ボスの出現時には、耐久力ゲージを0から最大値まで上げる演出を入れました。ボスの耐久力が大きいところをアピールして、プレイヤーに威圧感を与えることが目的です。耐久力ゲージの表示については後述します(P. 286)。なお、出現時のボスは破壊できません。

ボスの出現時にはスクリプトの進行を止めます。ボスと戦っている間は、ずっとスクリプトを止めておきます。これはボス以外の敵を出現させないためです。ボスとスクリプトの関係については後述します(P. 300)。

**Fig. 9-5** ボスの出現

出現時のボスは、時間の経過とともに動きが変化します。そのためにタイマーとなる変数をフレームごとに更新し、その値に応じて処理を変えます。

ボスは最初に警告メッセージのタスクを生成し、約1秒後に画面下方への移動を開始します。そして約1秒間かけて移動した後、約0.5秒待ってから、実際に攻撃を行うために第1段階のボス（CHig1）を生成します。

List 9-2は、出現時のボスに関するクラス（CHig）です。このクラスはボスの出現時に関する処理だけを行い、攻撃は後述する第1段階（P. 290）や第2段階（P. 294）のボスに相当するクラスで行います。

**List 9-2　ボスの出現（Enemy.h、Enemy.cpp）**

```
// ボス（出現時）のクラス
class CHig : public CEnemy {
public:

    // コンストラクタ、移動、描画、生成
    CHig(float x, float y=-80);
    virtual bool Move();
    virtual void Draw();

    // Chapter 8で解説した敵の出現処理に使う関数（P. 267）
    static CEnemy* New(float x) { return new CHig(x); }
};

// コンストラクタ
// 敵クラス（CEnemy）のコンストラクタを呼び出して
// 座標や当たり判定などを初期化
CHig::CHig(float x, float y)
:   CEnemy(Game->MeshHig, x, y,
        -11.0f, -11.0f, 11.0f, 11.0f, 0, 0)
{
    // スクリプトの停止
    Game->Script->Pause();
}

// 移動
bool CHig::Move() {

    // タイマーが0のとき：
    // 警告メッセージのタスクを生成し、耐久力ゲージを0にする
    if (Time==0) {
        new CWarning();
        Game->SetVitGauge(0);
    } else
```

```
    // タイマーが60以上120未満のとき：
    // 画面下方へ移動しつつ、ゲージを加算する
    if (60<=Time && Time<120) {
        Y++;
        Game->SetVitGauge((float)(Time-60)/60);
    } else

    // タイマーが150のとき：
    // ゲージを最大値にし、第1段階のボスを生成して、
    // 自分を消去する
    if (Time==150) {
        Game->SetVitGauge(1);
        new CHig1(X, Y);
        return false;
    }

    // タイマーの加算
    Time++;
    return true;
}

// 描画
// 他の敵に比べてボスは大きめに描く
void CHig::Draw() {
    DrawMesh(X, Z, -Y, 4, 4, 4, 0.125f, 0, 0, TO_ZYX, 1, false);
}
```

## 耐久力ゲージ

ボスは耐久力が大きいため、ショットやビームを当てても、どのくらい耐久力を削ったのかがわかりづらくなっています。そのため多くのゲームでは、ボスの耐久力をゲージで表示したり、破壊された跡をグラフィックで表示したりすることによって、ボスがダメージを受けたことを表現します。

ここでは、耐久力ゲージを画面上方に表示することにしました。List 9-3は、ゲージを描画するプログラムです。耐久力を表す変数（VitGauge）に0から1までの値を設定すると、値に応じた長さの矩形を描画し、耐久力をグラフ表示します。ゲージを表示する必要がないときには、耐久力を0にしておきます。そうするとゲージが表示されません。

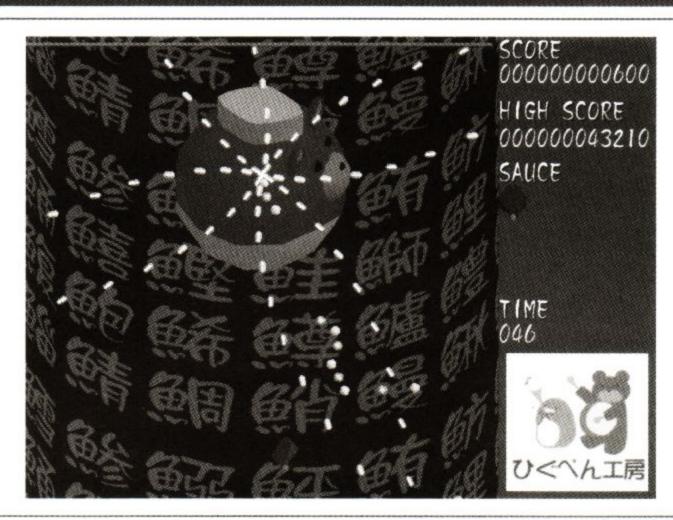
**List 9-3 耐久力ゲージの表示（Main.h、Main.cpp）**

```
// ゲーム本体のクラス
class CShtGame : public CGame {

    // ...（中略）...

    // 耐久力ゲージ（0.0～1.0）
    float VitGauge;
};

// 描画
void CShtGame::Draw() {

    // ...（中略）...

    // 耐久力ゲージの描画
    CTexture::DrawRect(
        device, 2, 10, h*VitGauge, 4,
        D3DCOLOR_ARGB(128, 64, 64, 64));
    CTexture::DrawRect(
        device, 0, 8, h*VitGauge, 4,
        D3DCOLOR_ARGB(255, 255, 0, 0));
}
```

## ボスの攻撃

さて、いよいよボスが攻撃を開始します。ボスは華麗かつ激しい攻撃でプレイヤーを魅了しなければならないので、攻撃パターンを念入りに作り上げる必要があります。

サンプルのボスは、ゆっくりと回転しながら多数の方向弾をばらまき、ときおり高速な狙い撃ち弾を発射する攻撃パターンを持っています（Fig. 9-6）。比較的シンプルなパターンですが、方向弾に加速度を与えて変化をつけました。発射直後の方向弾はゆっくり飛びますが、しだいにスピードが速くなります。

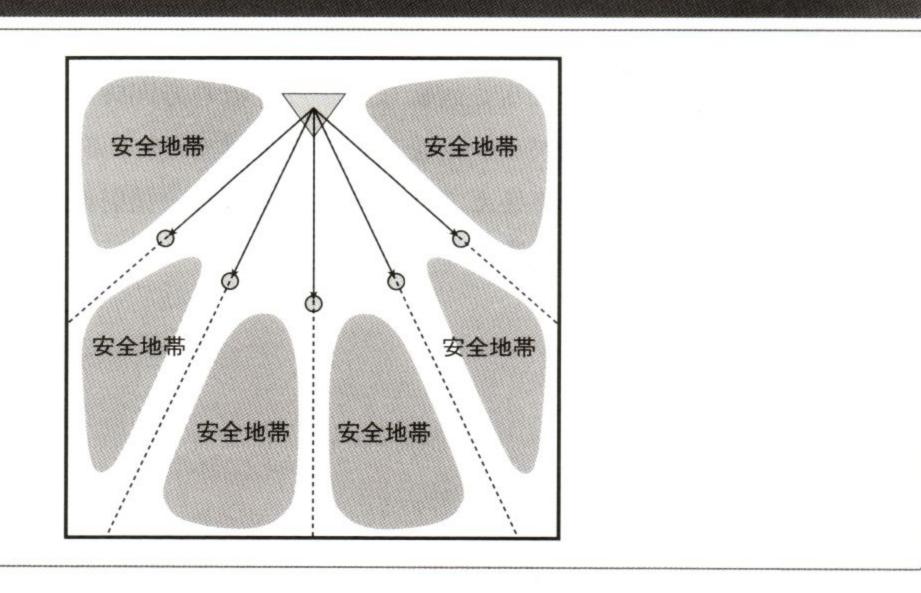
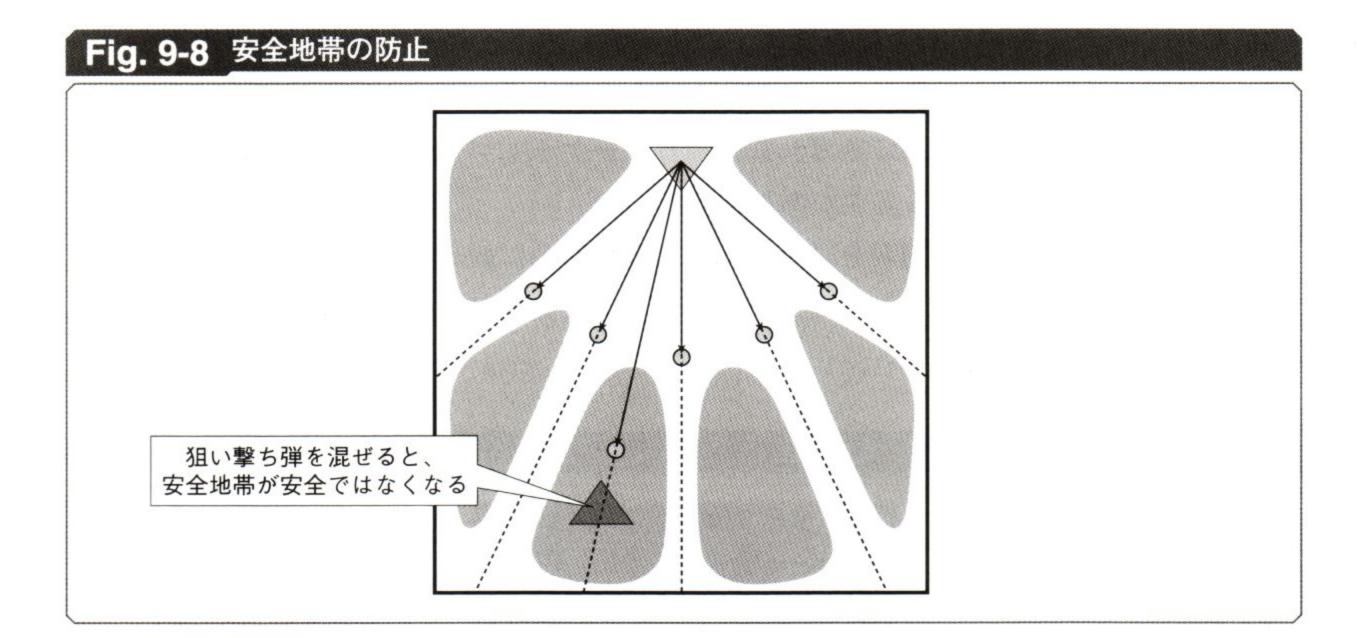
**Fig. 9-6** ボスの攻撃

## 安全地帯

規則的に密集した方向弾を発射すると、見た目に美しい弾幕を作ることができます。しかし、あまりに規則的にしてしまうと、自機を動かさないでも弾が避けられてしまうポイントができてしまいます。これを「安全地帯」または「安地（あんち）」と呼びます（Fig. 9-7）。

安全地帯ができないようにするには、不規則な弾を少し混ぜる方法があります。例えば、狙い撃ち弾を混ぜれば、自機を移動しなければ必ず被弾してしまうことになるので、安全地帯の発生を防ぐことができます（Fig. 9-8）。

本章のボスは方向弾に加えて、狙い撃ち弾を発射します。そのため、安全地帯はできません。

**Fig. 9-7** 安全地帯

Fig. 9-8 安全地帯の防止

## 時間制限

シューティングゲームを遊んでいると、ボスとの戦闘に時間制限が設けられている作品をよく見かけます。これはスコアを無限に稼ぎ続けたり、ゲームを無限に遊び続けたりすることへの対策です。

例えば、ボスが撃ってくるミサイルなどを破壊すると得点が入り、かつボス本体を破壊せずにいつまでも粘り続けることができる場合には、ボスとの戦闘で無限にスコアを稼ぐことができてしまいます。これは永久パターン（略して「永パ（えいパ）」などとも呼ばれます）の一種です。永久パターンとは、ある一連のプレイを繰り返すことによって、無限に得点が稼げるパターンのことです。

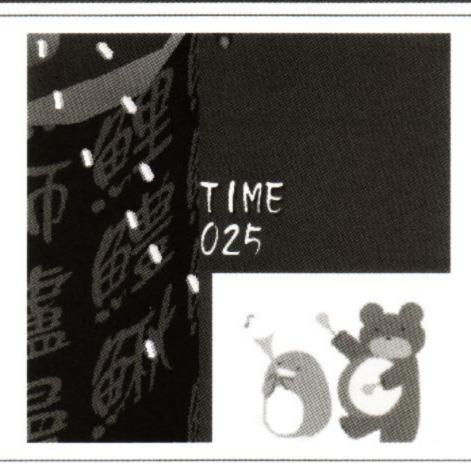
永久パターンが発見されてしまうと、そのゲームを普通にプレイして高スコアを目指す意味が失われてしまいます。これでは、ゲームを遊び込んでくれるプレイヤーが減ってしまうでしょう。そこで多くのゲームは、永久パターンを防止するためのなんらかの仕組みを取り入れています。

ボスに関しては、次のような対策があります。方法は違っても、ボスとの戦闘が無限に続かないようにするという目的は同じです。

- ボスを制限時間内に倒さないと、ボスが逃げるようにする
- ボスを一定時間内に倒さないとボスの攻撃が極端に厳しくなり、いつかは必ずミスするようにする
- 一定時間をすぎて粘っていると、プレイヤーにミスをさせるための特別なキャラクター（多くの場合は無敵のキャラクター）が出現する

PCゲームや家庭用ゲームの場合には、ボスが撃ってくるミサイルなどでは得点が入らないようにするだけでも十分です。ただしこの場合は、スコアは入らなくても無限に遊び続けることはできます。無限に遊ばれては困るアーケードゲームの場合には、制限時間を設けるなどの手法を併用する必要があります。

サンプルのボスには時間制限を設けて、規定の時間内に倒せなかった場合には、ボスが逃げるようにしました。残りの時間がわかるよう、画面の右側にあるスコア領域に残り時間を表示します（Fig. 9-9）。

**Fig. 9-9** 残り時間の表示

## 第1段階のボス

ボスには第1段階および第2段階があり、それぞれ異なる攻撃パターンを見せます。まず、第1段階のボスについて解説します。

最初にボスの当たり判定、耐久値、スコアなどを初期化します。耐久値は2000、スコアは10000としました。ちなみに通常の敵（ザコ）は、耐久値が10、スコアが100になっています。

また、戦闘の残り時間も設定します。時間は3600フレームです。このサンプルでは1フレームが1/60秒なので、残り時間は約60秒になります。

ボスは約1/6秒間隔で、時計回りと反時計回りに発射方向を変化させながら、方向弾を4発ずつ発射します。また、約0.5秒間隔で7発の狙い撃ち弾を、多少ランダムに方向をずらしつつ自機に向かって発射します。

ボスが破壊されるか、あるいは残り時間がなくなったら、ボスは第2形態に移行します。これは、第2形態のタスクを生成し、自分（第1形態のタスク）を消去することによって行います。破壊された場合には、爆発を生成し、スコアを加算します。

List 9-4は、第1段階のボスに相当するクラス（CHig1）です。

**List 9-4　ボスの攻撃（Enemy.h、Enemy.cpp）**

```
// ボス（第1段階）のクラス
class CHig1 : public CEnemy {
public:

    // コンストラクタ、移動、描画
    CHig1(float x, float y);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CHig1::CHig1(float x, float y)
:   CEnemy(
        Game->MeshHig, x, y,
        -11.0f, -11.0f, 11.0f, 11.0f, 2000, 10000)
{
    // 戦闘の残り時間を設定
    Game->SetBossTime(60*60);
}

// 移動
bool CHig1::Move() {
    CMyShip* myship=Game->GetMyShip();

    // 方向弾の発射：
    // 約1/6秒ごとに4発発射する
    // 発射方向はボスの回転と同じ方向へ回転していく
    if (Time%10==0) {
        for (int i=0, n=4; i<n; i++) {
            new CDirBullet(
                Game->MeshNeedle, Color,
                X, Y, Yaw+(float)i/n, 0.3f, 0.01f);
        }
    }

    // 方向弾の発射：
    // 約1/6秒ごとに4発発射する
    // 発射方向はボスの回転とは逆の方向へ回転していく
    if (Time%10==5) {
        for (int i=0, n=4; i<n; i++) {
            new CDirBullet(
                Game->MeshNeedle, 1-Color,
                X, Y, -Yaw+(i+0.5f)/n, 0.3f, 0.01f);
        }
    }
```

```
    // 狙い撃ち弾の発射：
    // 0.5秒間隔で発射と休止を繰り返す
    if (Time%60<30 && Time%4==0) {
        if (myship) {
            new CAimBullet(
                Game->MeshBullet, Color, X, Y,
                myship->X, myship->Y,
                Game->Rand05()*0.05f, 0.8f, 0);
        }
        Color=1-Color;
    }

    // 角度とタイマーの更新
    Yaw+=0.001f;
    Time++;

    // 耐久力ゲージの更新
    Game->SetVitGauge((float)Vit/2000);

    // 残り時間の更新
    int boss_time=Game->GetBossTime();
    Game->SetBossTime(boss_time-1);

    // 破壊されるか、あるいは一定時間が経過したとき：
    if (Vit<=0 || boss_time<0) {

        // 破壊されたときは爆発を生成し、スコアを加算する
        if (boss_time>=0) {
            new CBossCrash(X, Y);
            Game->AddScore(Score+boss_time*10);
        }

        // 第2形態の生成。ボスが2体に分裂する
        CHig2* left=new CHig2(X, Y, -1);
        CHig2* right=new CHig2(X, Y, 1);

        // 分裂した片割れへのポインタを設定
        left->SetFellow(right);
        right->SetFellow(left);

        return false;
    }
    return true;
}
```

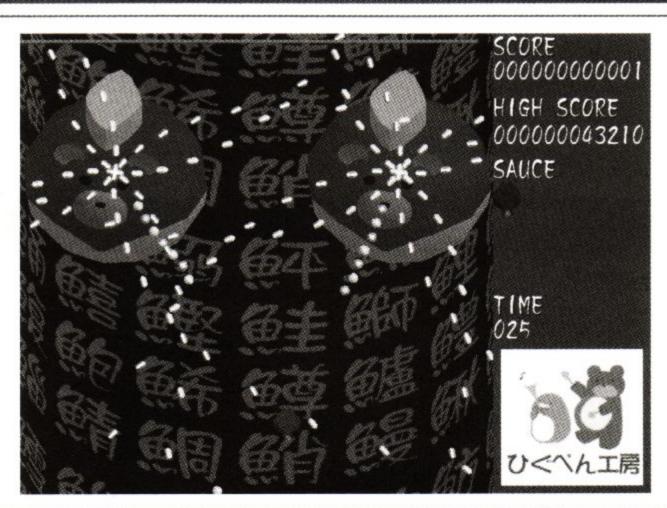
```
// 描画
void CHig1::Draw() {
    DrawMesh(X, Z, -Y, 4, 4, 4, 0.125f, Yaw, 0, TO_ZYX, 1, false);
}
```

# 攻撃パターンの変化

多くのゲームでは、特定の段階まで破壊されるごとに、ボスの攻撃パターンが変化します。ゲームによっては2段階や3段階、あるいはそれ以上にパターンが変化することもあります。また、攻撃パターンとともに、ボスが合体や分裂、あるいは変形などを行う場合も見受けられます。

ゲームでボスの攻撃パターンが変化するのは、アニメや特撮などで悪役が変身や変形をするのに近い雰囲気です。また、ボスの攻撃パターンがしだいに激しくなることによって、長いボス戦もあきさせずに楽しませることができます。

サンプルのボスにも、攻撃パターンの変化を取り入れてみました。ボスの第1段階を破壊すると、ボスが2体に分裂して攻撃がより激しくなります（Fig. 9-10）。それぞれのボスは第1段階とほぼ同じパターンで攻撃を行い、本体のグラフィックは変化しませんが、第1段階とはかなり雰囲気の違う攻撃になります。

**Fig. 9-10** 攻撃パターンの変化

## 第2段階のボス

第2段階のボスについて解説します。まず第2段階のボスは、分裂した片割れへのポインタを保持しています。このボスは2体に分裂しますが、耐久力は共有しています。そのため双方の情報を参照する必要があり、このようなポインタを使うのです。このポインタはボスの分裂時に設定します(List 9-4)。また、このようにボスのパーツが互いを参照できるようにしておくと、パーツを連携させて動かすといった処理も実現できます。

第2段階のボスは2体になるので、1体あたりの耐久力は第1段階の半分にしました。ボスの耐久力は2体分の合計値について表示し、合計値が0以下になったときに2体とも破壊されるようにします。

ボスの出現時には、2体のボスにそれぞれ異なる速度を設定することによって、2体を異なる方向(左または右)に移動させます。ボスは、最初の約0.5秒間は移動だけを行い、続いて攻撃を開始します。攻撃の内容は第1段階とほぼ同じですが、2体になったので弾は少なめにしました。

ボスが破壊されたとき、または時間切れになったときには、ステージクリア画面を生成し、第2段階のボスは消滅します。また、通常のステージ進行に戻るため、スクリプトを再開します。

ボスが破壊されたときには、スコアを加算し、爆発を生成します。時間切れのときには、退却時のボスを生成します。いずれの場合にも、分裂した片割れへのポインタを利用して、片割れを消去します。

なお、ボスの爆発時には画面上の弾を消す処理も行います。これは大きな敵の爆発時に弾を消す処理(P. 246)と同じです。この処理はボスの爆発クラス(CBossCrash)で行います。詳細は「Effect.h」と「Effect.cpp」をご覧ください。

List 9-5は、第2段階のボスに相当するクラス(CHig2)です。

**List 9-5　攻撃パターンの変化(Enemy.h、Enemy.cpp)**

```
// ボス(第2段階)のクラス
class CHig2 : public CEnemy {

    // X方向の速度
    float VX;

    // 分裂した片割れへのポインタ
    CHig2* Fellow;

public:
```

```
    // コンストラクタ、移動、描画
    CHig2(float x, float y, float vx);
    virtual bool Move();
    virtual void Draw();

    // 分裂した片割れへのポインタを設定する
    void SetFellow(CHig2* fellow) { Fellow=fellow; }
};

// コンストラクタ
CHig2::CHig2(float x, float y, float vx)
:   CEnemy(
        Game->MeshHig, x, y,
        -11.0f, -11.0f, 11.0f, 11.0f, 1000, 10000),
    VX(vx)
{
    // 戦闘の残り時間を設定
    Game->SetBossTime(60*60);
}

// 移動
bool CHig2::Move() {
    CMyShip* myship=Game->GetMyShip();

    // 分裂した片割れが消滅したら自分も消滅する
    if (!Fellow) return false;

    // タイマーが30以下のとき：
    // 左または右に移動する
    if (Time<30) {
        X+=VX;
    }

    // タイマーが30以上のとき：
    // 方向弾と狙い撃ち弾で攻撃を行う
    else {

        // 方向弾
        if (Time%16==0) {
            for (int i=0, n=4; i<n; i++) {
                new CDirBullet(
                    Game->MeshNeedle, Color, X, Y,
                    Yaw+(float)i/n, 0.2f, 0.005f);
            }
        }
```

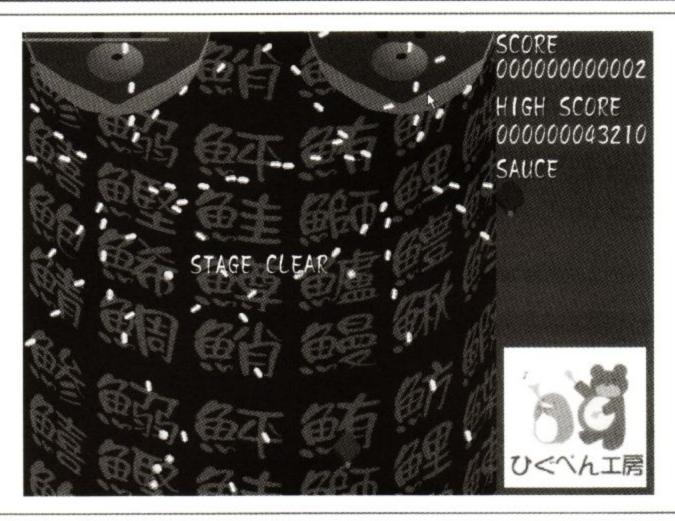
```
        // 方向弾
        if (Time%16==8) {
            for (int i=0, n=4; i<n; i++) {
                new CDirBullet(
                    Game->MeshNeedle, 1-Color,
                    X, Y, -Yaw+(i+0.5f)/n, 0.2f, 0.005f);
            }
        }

        // 狙い撃ち弾
        if (Time%90<30 && Time%4==0) {
            if (myship) {
                new CAimBullet(
                    Game->MeshBullet, Color, X, Y,
                    myship->X, myship->Y,
                    Game->Rand05()*0.01f, 0.8f, 0);
            }
            Color=1-Color;
        }
    }

    // 角度とタイマーの更新
    Yaw+=0.001f;
    Time++;

    // 耐久力ゲージの更新
    Game->SetVitGauge((float)(Vit+Fellow->Vit)/2000);

    // 残り時間の更新
    int boss_time=Game->GetBossTime();
    Game->SetBossTime(boss_time-1);

    // 破壊されたとき、または時間切れのとき
    if (Vit+Fellow->Vit<=0 || boss_time<0) {

        // ステージクリア画面の生成
        new CStageClear();

        // スクリプトの再開
        Game->Script->Resume();

        // 破壊されたときはスコアを加算して爆発を生成
        if (boss_time>=0) {
            Game->AddScore(Score+boss_time*10);
            new CBossCrash(X, Y);
```

```
                new CBossCrash(Fellow->X, Fellow->Y);
            }

            // 時間切れのときは退却時のボスを生成
            else {
                new CHig3(X, Y);
                new CHig3(Fellow->X, Fellow->Y);
            }

            // 残り時間を非表示にする
            Game->SetBossTime(-1);

            // 分裂した片割れに自分の消滅を知らせる
            Fellow->Fellow=NULL;
            return false;
        }
        return true;
}

// 描画
void CHig2::Draw() {
        DrawMesh(
            X, Z, -Y, 4, 4, 4,
            0.125f, Yaw, 0, TO_ZYX, 1, false);
}
```

## ボスの退却

時間切れになると、ボスは画面上方に逃げていきます（Fig. 9-11）。このボスは破壊できません。ボスは約1秒間だけ画面上方へ移動した後に、消滅します。

List 9-6は、退却するボスのクラス（CHig3）です。

**Fig. 9-11　ボスの退却**

**List 9-6　ボスの退却（Enemy.h、Enemy.cpp）**

```
// ボス（退却時）のクラス
class CHig3 : public CEnemy {
public:

    // コンストラクタ、移動、描画
    CHig3(float x, float y);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CHig3::CHig3(float x, float y)
:   CEnemy(
        Game->MeshHig, x, y,
        -11.0f, -11.0f, 11.0f, 11.0f, 0, 0)
{}

// 移動
// 上方向へ移動し、約1秒後に消滅
bool CHig3::Move() {
    Y--;
    Time++;
    return Time<60;
}

// 描画
void CHig3::Draw() {
    DrawMesh(X, Z, -Y, 4, 4, 4, 0.125f, 0, 0, TO_ZYX, 1, false);
}
```

# ステージクリア

ボスを倒すと、多くのゲームではステージクリアとなります。ゲームによっては、ステージをクリアすると、そのステージで獲得したスコアやボムの残り数、ボーナス、あるいはストーリーなどが表示されます。今回はシンプルに、Fig. 9-12のようなメッセージを表示することにしました。少しアレンジすれば、スコアやボーナス、あるいはストーリーやデモなども表示できます。

ステージクリア画面では、画面の中央付近にステージクリアのメッセージを表示します。そして約1.5秒後に、ボスの耐久力ゲージを消去して、自分（ステージクリア画面）も消滅します。

List 9-7は、ステージクリア画面を表すクラス（CStageClear）です。

**Fig. 9-12** ステージクリア

**List 9-7** ステージクリア（Scene.h、Scene.cpp）

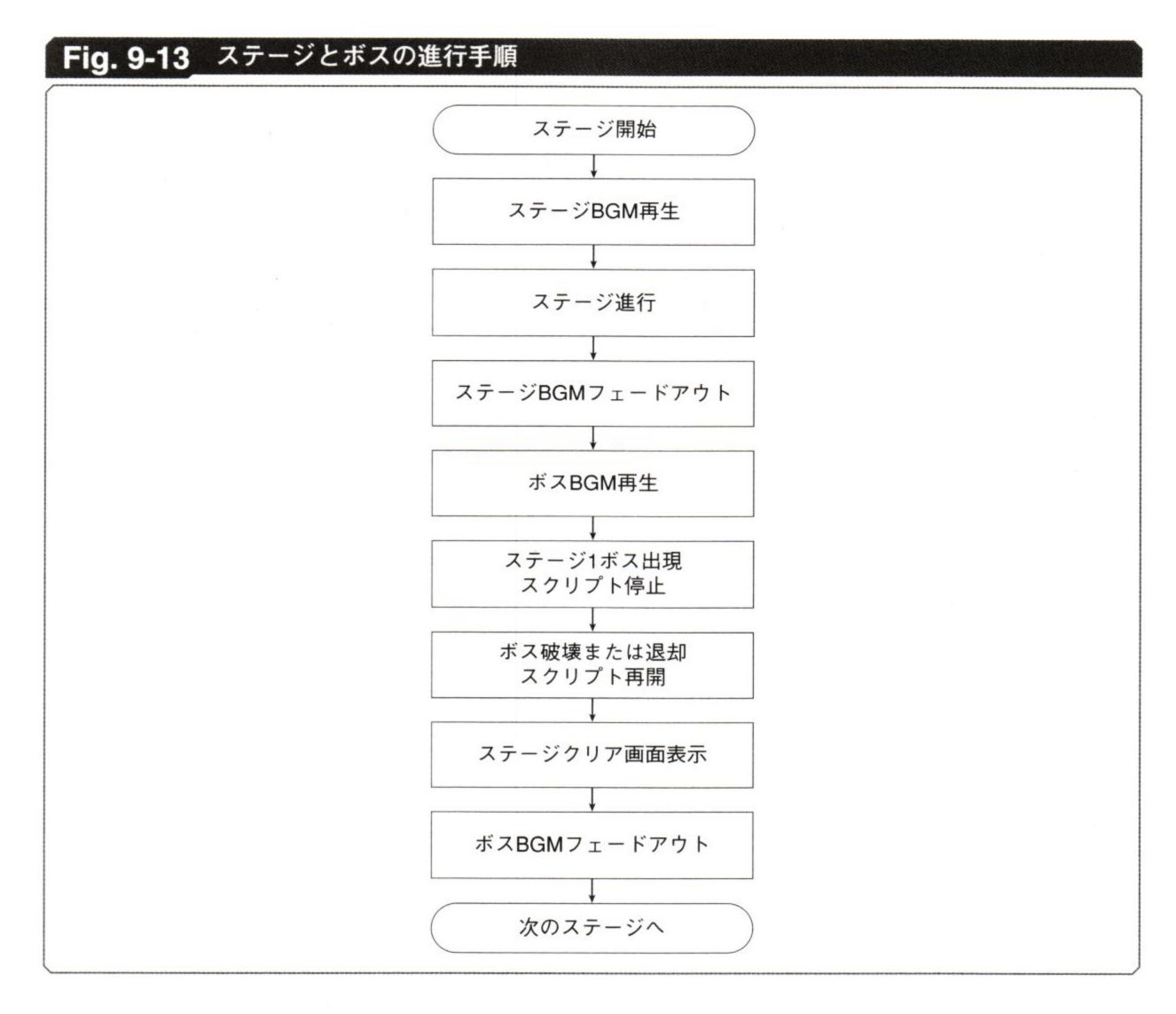
```
// ステージクリア
// 各種の画面を表すクラス（CScene）から派生
class CStageClear : public CScene {
protected:

    // タイマー
    int Time;

public:
```

```
    // コンストラクタ、移動、描画
    CStageClear();
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CStageClear::CStageClear()
:   Time(0)
{}

// 移動
bool CStageClear::Move() {

    // タイマーが90を超えたとき：
    // ボスの耐久力ゲージを消去し、自分も消滅する
    Time++;
    if (Time>90) {
        Game->SetVitGauge(0);
        return false;
    }
    return true;
}

// 描画
void CStageClear::Draw() {
    int h=Game->GetGraphics()->GetHeight();
    Game->Font->DrawText(
        "STAGE CLEAR", h/2-11*8, h/2-16, ColWhite, ColShade);
}
```

## ステージの進行とボス

多くのシューティングゲームでは、ステージが始まると最初は通常の敵が出現します。ステージの最後にはボスが出現して、ボスを破壊するかまたはボスが退却すると、ステージクリアとなります。そして、得点やメッセージなどを表示した後に、次のステージへ進みます。通常のステージ進行時とボスとの戦闘時では、BGMも切り替わるのが一般的です。

通常のステージとボスに関する進行の手順を図にまとめました(Fig. 9-13)。

**Fig. 9-13 ステージとボスの進行手順**

### ステージ開始

ステージ用のBGMを再生し、ステージを開始します。

### ステージ進行

スクリプトを使って敵を生成し、ステージを進行します。

### ボス開始

ステージ用のBGMをフェードアウトさせ、ボス用のBGMに切り替えます。そしてボスを生成し、スクリプトを停止させます。スクリプトを停止させるのは、ボスとの戦闘が終わるまで他の敵の生成を止めるためです。

### ボス終了

ボスが破壊されるか、あるいはボスが退却したら、戦闘は終了です。スクリプトを再開

して、通常のステージ進行に戻ります。また、ボス用のBGMをフェードアウトさせます。ステージの最後に登場するボスの場合には、ステージクリア画面を表示します。

### 次のステージへ

ステージ開始時と同様の手順で、次のステージを開始します。

## ボスを含むスクリプト

List 9-8は、ボスを含むスクリプトの例です。Fig. 9-13のような手順で、ステージを進行し、ボスを出現させます。ボスを含まないスクリプト（P. 258）と見比べてみてください。

このスクリプトでは、最初にステージ1用のBGMを再生します。そして通常の敵を出現させ、ステージ1を進行します。

ボスの出現が近づいたら、出現に備えてステージ1のBGMをフェードアウトさせます。そして、ステージ1のBGMにかわって、ボスのBGMを再生します。

次にボスを出現させます。ボスの出現時には、出現時のボスを表すクラス（CHig）のなかでスクリプトを停止します。ボスの破壊時または退却時には、第2段階のボスを表すクラス（CHig2）のなかでスクリプトを再開します。

ボスとの戦闘が終了したら、ボスのBGMをフェードアウトさせます。そしてステージ2用のBGMを再生し、ステージ2を開始します。続いて通常の敵を出現させ、ステージ2を進行します。以下のステージについても手順は同様です。

ボスの警告メッセージやステージクリア画面などは、ボスを表すクラスのなかからインスタンスを生成していますが、これらをスクリプトで生成する方法もあります。どちらでも、プログラムが書きやすい方法を選べばよいでしょう。

**List 9-8　ボスを含むスクリプト（script.txt）**

```
// ステージ1BGM再生
play 0
wait 60

// ステージ1進行
enemy akami 10
wait 10
enemy akami 20
wait 10
enemy akami 30
wait 10
enemy akami 40
```

```
wait 100

// ...(中略)...

// ステージ1BGMフェードアウト
fadeout 100
wait 100

// ボスBGM再生
play 4

// ボス出現(スクリプト停止、ボス終了後再開)
enemy hig 0

// ボスBGMフェードアウト
fadeout 100
wait 100

// ステージ2BGM再生
play 1
wait 60

// ステージ2進行
enemy akami 10
wait 10
enemy akami 20
wait 10
enemy akami 30
wait 10
enemy akami 40
wait 100

// ...(中略)...
```

## スクリプトの停止と再開

スクリプトの進行を停止したり再開したりする処理は、List 9-9のように実現しています。Chapter 8で作成したスクリプトクラス(CScript)を拡張しました(P. 265)。

Chapter 8で実装したスクリプトの実行処理(Run関数)では、スクリプトの待ち時間(Wait)が0以外のときにはコマンドを実行しないようにしていました。そのため、例えば待ち時間を-1にすれば、スクリプトの実行を止めることができます。

スクリプトの停止と再開にはこれを利用しました。スクリプトクラスに、スクリプトの実行を一時停止するPause関数と、再開するResume関数を追加します。それぞれ、待ち時

間を表すメンバ変数Waitに対して、-1および0を代入します。

**List 9-9　スクリプトの停止と再開（Script.h、Script.cpp）**

```
// スクリプト
class CScript {
    vector<CCommand*> Command;
    int CommandIndex, Wait;
public:
    CScript(string file);
    void Init();
    void Run();
    void SetWait(int wait) { Wait=wait; }

    // 一時停止と再開
    void Pause() { Wait=-1; }
    void Resume() { Wait=0; }
};

// スクリプトの実行
void CScript::Run() {
    if (Wait>0) Wait--;

    // Waitが0以外のときはコマンドを実行しない
    while (CommandIndex<(int)Command.size() && Wait==0) {
        Command[CommandIndex]->Run();
        CommandIndex++;
    }
}
```

## >>Chapter 9のまとめ

本章ではボスの出現や移動、攻撃パターンの変化などについて解説しました。ボスはゲームを盛り上げてくれる要素ですが、必ずゲームに組み込まなければならないというわけでもありません。むやみにボスを多用すると、かえって退屈なゲームになってしまう危険性もあります。ボス戦を導入する場合には、登場のタイミングや攻撃パターン、耐久力などを調整して、楽しく倒せるボスを作りましょう。

次章ではアイテムをテーマに、パワーアップアイテム、得点アイテム、敵や弾のアイテムへの変化、アイテムの自動回収などについて説明します。

# Chapter 10

# アイテムとパワーアップ

本章のテーマはアイテムとパワーアップです。アイテムとパワーアップがあると、プレイヤーが敵を倒す動機づけが強くなります。また、ミスをしたときにパワーダウンするルールを設定すると、プレイヤーはミスをせずに上手にプレイしようという気持ちになります。
本章では、アイテムの出現と自動回収、得点アイテム、パワーアップアイテム、味方機、アイテムの放出などについて説明します。

Shooting Game Programming

# アイテムとは

多くのシューティングゲームにはアイテムが出現します。代表的なアイテムには次のようなものがあります。

- 得点アイテム
- パワーアップアイテム
- ボムアイテム

得点アイテムは、取るとスコアが増えるアイテムです。最近のシューティングゲームでは、画面をうめつくすほどの得点アイテムが出現することもあります。膨大なアイテムが出現するゲームの例としては「ギガウィング2」(アーケード/ドリームキャスト)などがあげられます。

パワーアップアイテムは、取ると自機のショットやビームが強くなったり、味方機(オプション)がついたりするアイテムです。ゲームによっては、アイテムによって武器の種類を切り替えられることもあります。

ボムアイテムは、取るとボムのストックが増えるアイテムです。パワーアップアイテムやボムアイテムは、パワーが最高段階のときやボムの所持数が最大に達しているときに取ると、一般にスコアが加算されます。

本章ではこういったアイテムについて解説します。本章のプロジェクトは「ShtGame_Item」フォルダに収録しています。実行ファイルは「ShtGame_Item¥Release¥ShtGame.exe」です。

# アイテムの出現

アイテムの出現方法には、主に次のようなパターンがあります。

- 敵を破壊すると出現する
- 輸送機を破壊すると出現する
- 弾や敵がアイテムに変化する
- ミスをしたときに自機からパワーアップアイテムが出現する

得点アイテムは敵を破壊したときに、パワーアップアイテムは輸送機を破壊したときに出現するゲームが多いようです。ゲームによっては、ボムを使用したときや特定の大きな

敵を倒したときなどに、画面上の弾や敵が得点アイテムに変化することもあります。また、ミスをしたときに再パワーアップ用のアイテムを自機が放出するゲームもよく見られます。

ここでは、敵を倒したときにアイテムを出現させることにしました。小さなザコ敵を倒したときには得点アイテムを1個だけ、大きな敵を倒したときには10個の得点アイテムと1個のパワーアップアイテムを出現させます。一方、ボスを破壊したときには、画面上にあるすべての弾が得点アイテムに変化します（Fig. 10-1）。

サンプルで出現するアイテムの種類と効果は次のとおりです。ボムアイテムについてはChapter 11で解説します（P. 344）。

Fig. 10-1 敵を破壊するとアイテムが出現

### 得点アイテム

寿司桶の形をしています。取るとスコアが入ります。

### パワーアップアイテム

醤油入れの形をしています。取ると自機に味方機がついて攻撃力が上がります。

## アイテムに関するクラス

Fig. 10-2はアイテムに関するクラス構成です。各アイテムのクラスを新規に作成する他、爆発を表すクラスの機能を拡張します。また、アイテムの取得時に得点のテキストを表示するクラスも作成します。各クラスの役割は次のとおりです。

### CItem

アイテムの基本機能をまとめたクラスです。移動物体を表すCMoverクラスから派生します。

### CScoreItem

得点アイテムです。CItemクラスから派生します。

### CPowerUpItem

パワーアップアイテムです。CItemクラスから派生します。

### CEnemyCrash

敵の爆発です。得点アイテムを生成する処理を追加します。

### CBigEnemyCrash

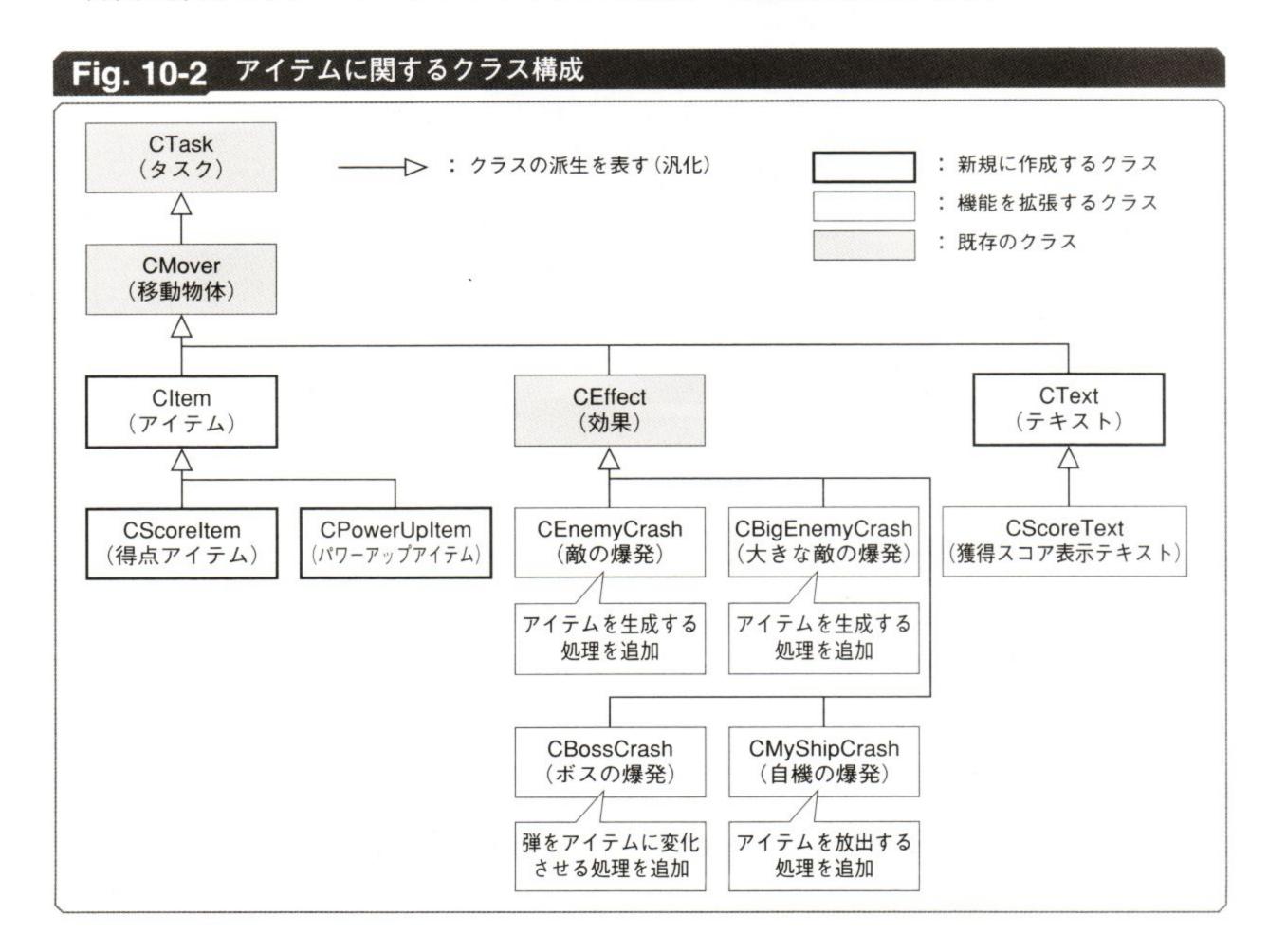
大きな敵の爆発です。得点アイテムとパワーアップアイテムを生成する処理を追加します。

### CBossCrash

ボスの爆発です。画面上の弾を得点アイテムに変化させる処理を追加します。

### CMyShipCrash

自機の爆発です。パワーアップアイテムを放出する処理を追加します。

**Fig. 10-2** アイテムに関するクラス構成

### CText

テキストを表示するための基本機能をまとめたクラスです。CMoverクラスから派生します。

### CScoreText

獲得したスコアのテキストを表示するためのクラスです。CTextクラスから派生します。アイテムを取ったときに獲得スコアを表示するために使います。

## アイテムの生成

敵を倒したときにアイテムを出現させるには、例えば、敵の爆発を初期化する際にアイテムも生成するようにします。倒した敵と同じ座標にアイテムを生成すれば、破壊した敵からアイテムが出たように見えるというわけです。

小さな敵を倒したときには、1個の得点アイテムを生成します。また、大きな敵を倒したときには、複数の得点アイテムと、パワーアップアイテムを生成します。アイテムの生成は、各アイテムを表すクラスのインスタンスを生成することによって行います。

複数の得点アイテムを生成するときには、まったく同じ座標に生成するのではなく、乱数を使うなどして位置を少しずらすとよいでしょう。まったく同じ座標に生成するとアイテム同士が重なってしまい、複数のアイテムが出たことがよくわからないからです。

List 10-1は、アイテムを出現させるプログラムです。

**List 10-1 アイテムの出現(Effect.cpp)**

```
// 敵の爆発を表すクラスのコンストラクタ
CEnemyCrash::CEnemyCrash(float x, float y, float scale)
:   CEffect(x, y), Time(0), Scale(scale)
{
    // 得点アイテムの生成
    new CScoreItem(x, y);

    // 爆発音の再生
    Game->PlaySE(Game->SECrash);
}

// 大きな敵の爆発を表すクラスのコンストラクタ
CBigEnemyCrash::CBigEnemyCrash(float x, float y)
:   CEnemyCrash(x, y, 0.6f)
{
    // 得点アイテムの生成
    for (int i=0; i<10; i++) {
```

```
        new CScoreItem(
            x+Game->Rand05()*20, y+Game->Rand05()*20);
    }

    // パワーアップアイテムの生成
    CMyShip* myship=Game->GetMyShip();
    if (myship) {
        new CPowerUpItem(x, y);
    }

    // 弾の消去
    for (CTaskIter i(Game->BulletList); i.HasNext(); ) {
        CBullet* bullet=(CBullet*)i.Next();
        float dx=x-bullet->X, dy=y-bullet->Y;
        if (dx*dx+dy*dy<1000) {
            bullet->Crash();
            i.Remove();
        }
    }

    // 爆発音の再生
    Game->PlaySE(Game->SEBigCrash);
}
```

## 弾をアイテムに変化させる

ボスを破壊したときには、画面上のすべての弾を得点アイテムに変化させます。このように弾をアイテムに変えるには、すべて弾を消去するのと同時に、それぞれの弾と同じ座標にアイテムを出現させます。これで、弾がアイテムに変化したように見えます。

List 10-2は、弾をアイテムに変化させるプログラムです。

**List 10-2 弾をアイテムに変化させる (Effect.cpp)**

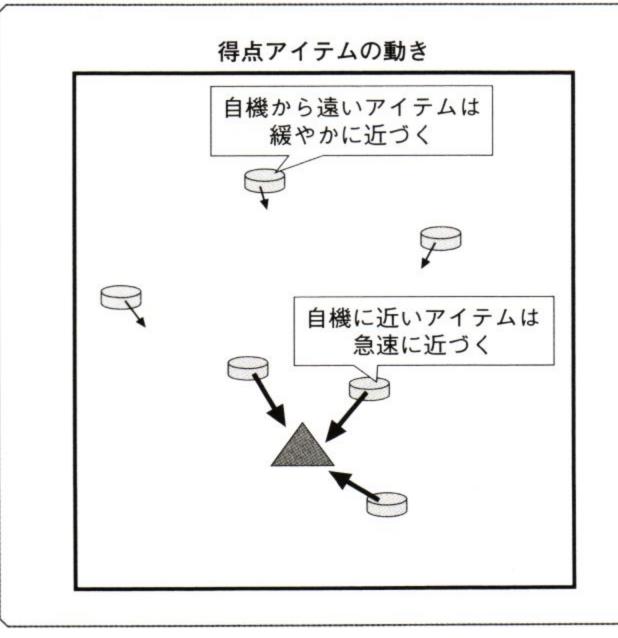
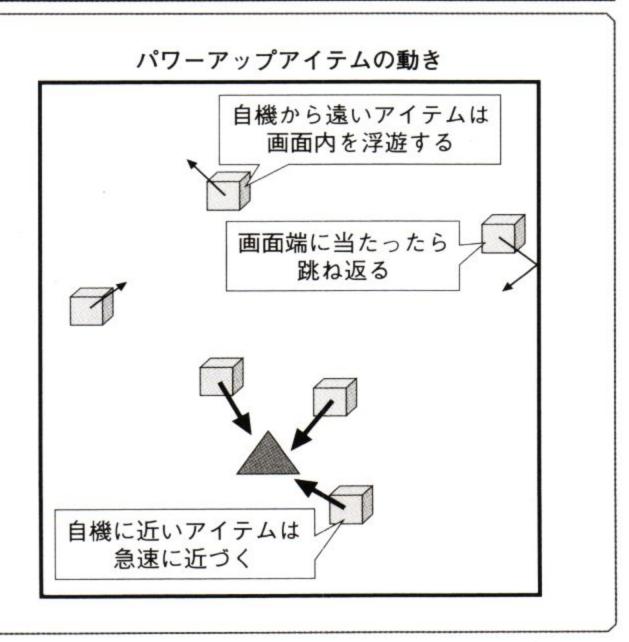
```
// ボスの爆発を表すクラスのコンストラクタ
CBossCrash::CBossCrash(float x, float y)
:   CEnemyCrash(x, y, 2.0f)
{
    // すべての弾を消去し、同じ座標に得点アイテムを生成する
    for (CTaskIter i(Game->BulletList);
        i.HasNext(); i.Remove()
    ) {
        CBullet* bullet=(CBullet*)i.Next();
```

```
        // 得点アイテムの生成
        new CScoreItem(bullet->X, bullet->Y);

        // 弾の破壊
        bullet->Crash();
    }

    // 爆発音の再生
    Game->PlaySE(Game->SEBossCrash);
}
```

# アイテムの自動回収

昔のゲームでは自機をアイテムにぴったり重ねないと取れないことが多かったのですが、最近のゲームでは、自機をアイテムにある程度近づければ、あとはアイテムが自動的に自機へ近づいてきて回収することができます。これがアイテムの自動回収です。自動回収があると、アイテムの回収がぐっと楽になり、取りこぼしも少なくなるので、ストレスなく快適にゲームを遊ぶことができます。

アイテムの自動回収にはいろいろな方式がありますが、ここではFig. 10-3のような例を紹介します。得点アイテムとパワーアップアイテムは少し違った動きをすることにしました。

**Fig. 10-3** 自動回収の動き

得点アイテムの場合、アイテムと自機が遠いときには、アイテムは緩やかな速度で自機に近づいてきます。自機がアイテムに一定距離まで近づくと、アイテムは急速に自機へと近づき、回収されます。

パワーアップアイテムの場合、アイテムと自機が遠いときには、アイテムは画面端で跳ね返りながら画面内を浮遊します。自機がアイテムに一定距離まで近づいたときには、得点アイテムと同様に自機へ急速に近づき、回収されます。

## アイテムの基本機能

得点アイテムとパワーアップアイテムには共通の機能があります。こういったアイテムの基本機能は1つのクラスにまとめて、そこから各種のアイテムを表すクラスを派生させるとよいでしょう。

アイテムの基本機能としては、まず当たり判定が必要です。自機がアイテムを回収するには、自機とアイテムの間で当たり判定処理を行わなければなりません。そのためには、アイテムに当たり判定を設定しておく必要があります。弾や敵とは違って、自機とアイテムはある程度接触しやすい方がよいので、当たり判定は大きめに設定します。

List 10-3は、アイテムの基本機能に関するプログラムです。

**List 10-3　アイテムの基本機能（Item.h）**

```
// アイテムの基本機能をまとめたクラス
class CItem : public CMover {
protected:

    // 回転角度
    float Yaw;

public:

    // new演算子、delete演算子
    // タスクリストにはアイテムタスクリスト(ItemList)を指定
    void* operator new(size_t t) {
        return operator_new(t, Game->ItemList);
    }
    void operator delete(void* p) {
        operator_delete(p, Game->ItemList);
    }
```

```
    // コンストラクタ
    // 「-3, -3, 3, 3」が当たり判定を表す(6x6の矩形)
    CItem(float x, float y)
    :   CMover(Game->ItemList, x, y, ITEM_Z, -3, -3, 3, 3),
        Yaw(0)
    {}
};
```

## 得点アイテムの動作

得点アイテムのポイントは、自動回収の処理です。自動回収を行うには、まずアイテムから自機へのベクトルと距離を求めます。次に、距離に応じて異なる速さで自機に接近します。自機から遠いときは遅く、自機に近いときには速く接近することにしました。

続いて、自機との当たり判定処理を行います。自機に接触したときには、スコアを加算してから消滅します。こうすることによって、アイテムを取るとアイテムが消え、スコアが入ります。

List 10-4は、得点アイテムに関するプログラムです。

**List 10-4 得点アイテム(Item.h、Item.cpp)**

```
// 得点アイテムのクラス
// アイテムの基本クラス(CItem)から派生
class CScoreItem : public CItem {
public:
    CScoreItem(float x, float y);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CScoreItem::CScoreItem(float x, float y)
:   CItem(x, y)
{}

// 移動
bool CScoreItem::Move() {
    CMyShip* myship=Game->GetMyShip();

    // 自動回収の距離、自動回収時の速さ(速い、遅い)
    static const float
        DIST=20, SPD_FAST=1.2f, SPD_SLOW=0.5f;

    // 回転角度の更新
    Yaw+=0.01f;
```

```
    // アイテム自動回収の処理
    if (myship) {

        // 自機に向かうベクトルと距離を求める
        float vx, vy, l;
        vx=(myship->X-X);
        vy=(myship->Y-Y);
        l=sqrt(vx*vx+vy*vy);

        // 距離に応じて異なる動きをする
        if (l>0) {

            // 自機に近いときは急速に接近する
            if (l<DIST) {
                X+=vx/l*SPD_FAST;
                Y+=vy/l*SPD_FAST;
            }

            // 自機から遠いときには緩やかに接近する
            else {
                X+=vx/l*SPD_SLOW;
                Y+=vy/l*SPD_SLOW;
            }
        }

        // 自機に接触したとき:
        // スコアを増やしてから消滅する
        if (Hit(myship)) {
            Game->PlaySE(Game->SEScoreItem);
            Game->AddScore(10);

            // スコアのテキストを表示
            new CScoreText(X, Y, 10);

            return false;
        }
    }

    // 自機が画面上にないときには下方向に進む
    else {
        Y+=SPD_SLOW;
    }

    return true;
}
```

```
// 描画
// 小さな寿司桶の3Dモデルを表示
void CScoreItem::Draw() {
    Game->MeshOkeSmall->Draw(
        X, Z, -Y, 3, 3, 3, 0.125f, Yaw, 0, TO_ZYX, 1, false);
}
```

## パワーアップアイテムの動作

パワーアップアイテムと得点アイテムでは、自機から遠いときの動きが違います。また、パワーアップアイテムが自機に接触したときには得点を増やすのではなく、味方機（オプション）の数を増やします。味方機は自機の周囲を飛んで、攻撃を支援します（P. 320）。

パワーアップアイテムは斜めに直進して、画面端に当たったときには跳ね返ります。通常時は一定速度で進み、画面外に出そうになったときに速度の方向を反転させれば、この動きが実現できます。なお、アイテムの出現時にはランダムに進行方向を決めます。

アイテムを自動回収する仕組みは、得点アイテムの場合と同様です。ただし、得点アイテムは自機から遠いときにも少しずつ自機に接近しますが、パワーアップアイテムは自機とは無関係に直進します。また、自機に接触したときには味方機を増やしてから消滅します。

List 10-5は、パワーアップアイテムに関するプログラムです。

**List 10-5　パワーアップアイテム（Item.h、Item.cpp）**

```
// パワーアップアイテムのクラス
// アイテムの基本クラス（CItem）から派生
class CPowerUpItem : public CItem {
protected:

    // 速度のX成分、Y成分
    float VX, VY;

public:

    // コンストラクタ、移動、描画
    CPowerUpItem(float x, float y);
    virtual bool Move();
    virtual void Draw();

};

// コンストラクタ
```

```
CPowerUpItem::CPowerUpItem(float x, float y)
:   CItem(x, y), VX(0.3f), VY(0.3f)
{
    // 最初の移動方向をランダムに決める
    // （斜め左上・左下・右上・右下の4種類）
    if (Game->Rand05()<0) VX=-VX;
    if (Game->Rand05()<0) VY=-VY;
}

// 移動
bool CPowerUpItem::Move() {
    CMyShip* myship=Game->GetMyShip();

    // 自動回収の距離、自動回収時の速さ
    static const float
        DIST=20, SPD=1.2f;

    // 画面端に当たったら跳ね返る
    if (X<-45 && VX<0 || X>=45 && VX>0) VX=-VX;
    if (Y<-45 && VY<0 || Y>=45 && VY>0) VY=-VY;

    // 回転角度の更新
    Yaw+=0.01f;

    // アイテム自動回収の処理
    if (myship) {

        // 自機に向かうベクトルと距離を求める
        float vx, vy, l;
        vx=(myship->X-X);
        vy=(myship->Y-Y);
        l=sqrt(vx*vx+vy*vy);

        // 自機に近いときは接近する
        if (l>0 && l<DIST) {
            X+=vx/l*SPD;
            Y+=vy/l*SPD;
        }

        // 自機から遠いときには直進する
        else {
            X+=VX;
            Y+=VY;
        }

        // 自機に接触したとき：
```

```
            // 味方機を増やしてから消滅する
            if (Hit(myship)) {
                Game->PlaySE(Game->SEPowerItem);
                myship->AddOption();
                return false;
            }
        }

        // 自機が画面上にないときには直進する
        else {
            X+=VX;
            Y+=VY;
        }

        return true;
}

// 描画
// 小さな醤油ビンの3Dモデルを表示
void CPowerUpItem::Draw() {
    Game->MeshSauce->Draw(
        X, Z, -Y, 0.6f, 0.6f, 0.6f,
        0.125f, Yaw, 0, TO_ZYX, 1, false);
}
```

## アイテムを取ったときに獲得したスコアを表示する

最近のゲームには、アイテムを取ったときや敵を倒したときに、ゲーム画面上に小さな文字で獲得したスコアを表示するものがあります。これはスコアを得たことを強調して、プレイヤーを楽しませることが目的です。連続してスコアを獲得すると、ゲーム画面が100や200といったスコア表示で埋め尽くされるので、見た目にも楽しく気分が盛り上がります。

ここでは、アイテムを取ったときに獲得スコアを表示してみました（Fig. 10-4）。同様の方法で、敵を倒したときなどに獲得スコアを表示することもできます。

アイテムを取得したときに、そのアイテムの位置に獲得スコアを表示すれば、このような獲得スコア表示が実現できます。タスクシステムを使う場合には、獲得スコア表示を表すタスクを、アイテムの位置に生成すればよいでしょう。スコアの表示にはChapter 2のフォントクラスが使えます（P. 35）。

獲得スコア表示は一定時間が経過したら消去しておくと、ゲーム画面を見やすく保てま

す。このとき、アルファ値を時間とともに減少させれば、スコア表示をなめらかに消去することができます。

本書のサンプルのように、ゲーム画面の描画には3Dグラフィックを使い、文字の描画に2Dグラフィックを使う場合には、座標変換に注意が必要です。一般に、3Dグラフィックの場合はあらかじめ設定した行列によって座標変換が行われますが、2Dグラフィックの場合は座標変換がなく、画面上の座標を直接指定します。そのため、3Dグラフィックと2Dグラフィックを組み合わせて表示するときには、両者の位置関係が正しくなるように、プログラムで座標変換を行う必要があります。

座標変換の詳細は、座標系の設定方法によって異なります。本書の場合には、ゲーム画面のX座標とY座標はともに-50から50の範囲です。2Dグラフィックで文字を表示するときには、これを画面上の座標に変換してから描画しています。

List 10-6は、獲得スコア表示のプログラムです。テキスト表示の基本機能をまとめたクラス（CText）と、獲得スコア表示のクラス（CScoreText）を作成しました。アイテム取得時に（List 10-4）、この獲得スコア表示クラスのインスタンスを作成することによって、ゲーム画面に獲得スコアを表示します。

**Fig. 10-4 アイテム取得時の獲得スコア表示**

**List 10-6 獲得スコア表示（Item.h、Item.cpp）**

```
// テキスト表示の基本機能をまとめたクラス
class CText : public CMover {
public:

    // new演算子、delete演算子
    // タスクリストはテキストタスクリスト(TextList)を指定
```

```
    void* operator new(size_t t) {
        return operator_new(t, Game->TextList);
    }
    void operator delete(void* p) {
        operator_delete(p, Game->TextList);
    }

    // コンストラクタ
    CText::CText(float x, float y)
    :   CMover(Game->TextList, x, y, 0)
    {}
};

// 獲得スコア表示のクラス
class CScoreText : public CText {
protected:

    // 獲得スコアの文字列
    char Score[20];

    // タイマー、文字列の長さ
    int Time, StrLen;

public:

    // コンストラクタ、移動、描画
    CScoreText(float x, float y, int score);
    virtual bool Move();
    virtual void Draw();
};

// コンストラクタ
CScoreText::CScoreText(float x, float y, int score)
:   CText(x, y), Time(60)
{
    // 獲得スコアを数値から文字列に変換
    itoa(score, Score, 10);

    // 文字数を取得
    StrLen=(int)strlen(Score);
}

// 移動
// 一定時間が経過したら消去
bool CScoreText::Move() {
    Time--;
```

```
    return Time>0;
}

// 描画
// 時間とともにアルファ値を減少させながら獲得スコアを表示
// 文字数に応じて表示位置を調整
void CScoreText::Draw() {
    float h=Game->GetGraphics()->GetHeight();
    Game->Font->DrawText(
        Score,
        (X+50)*0.01f*h-8*StrLen, (Y+50)*0.01f*h-16,
        D3DXCOLOR(1, 1, 1, Time/60.0f));
}
```

## 味方機

パワーアップアイテムを取ると味方機（オプション）が増えて、自機の攻撃力が上がるようにしました。この味方機は、自機がショットやビームを発射するのに合わせてショットを撃ちます。また、ショットのときには広範囲への攻撃を行うように、ビームのときには前方に集中した攻撃を行うように、隊列を変化させます（Fig. 10-5）。

隊列をなめらかに変化させることが、味方機の動きをきれいに見せるためのポイントです。最近のゲームでは、「虫姫さま」（アーケード/PS2）や「グラディウスV」（PS2）などが参考になるでしょう。

**Fig. 10-5　味方機によるパワーアップ**

ショット時の味方機

ビーム時の味方機

## 味方機の移動

味方機の隊列はショット時とビーム時で異なります。ショット時には、広めに間隔を取って、自機の背後に味方機を横1列に並べます。ビーム時には、自機の背後に味方機を密集させます。

隊列をなめらかに変化させるには、味方機を一瞬で目標の位置に移動させるのではなく、目的の位置に向かって少しずつ移動させることがポイントです。そのためには、味方機の座標から移動先の座標へのベクトルと距離を求めます。そして、距離が近い場合には指定座標に直接移動し、遠い場合には決められた速さで指定座標に近づくようにします。移動の速さに上限を設けることで、味方機の隊列をなめらかに変化させることができます。

味方機の初期座標は、例えば自機と同じ座標にしておくとよいでしょう。また、味方機は増減するので、変数で味方機の数を管理する必要があります。

味方機の数に応じて、自機の後方に味方機を描画します。サンプルでは、自機と同じ醤油ビンの3Dモデルをやや小さめに描画することで、味方機を表現しています。

List 10-7は、味方機の移動に関するプログラムです。自機の基本機能に相当するクラス(CMyShip)の内部で、味方機のクラス(COption)を定義しました。自機と味方機は密接に関連しているので、このように内部クラスにしておくと、自機から味方機のクラスを自由に操作できて便利です。もちろん、内部クラスにせずに外部でクラスを定義してもかまいません。

**List 10-7　味方機の移動(MyShip.h、MyShip.cpp)**

```
// 自機の基本クラス
class CMyShip : public CMover {

    // ...(中略)...

    // 味方機の上限数
    enum {
        MAX_OPTIONS=4
    };

    // 味方機のクラス
    class COption {

        // 座標
        float X, Y, Z;

    public:
```

```
        // 座標の設定
        void SetXY(float x, float y) { X=x; Y=y; }

        // 指定座標への移動
        void MoveTo(float x, float y);

        // 描画
        void Draw(float roll, float alpha);

        // ショットの発射
        void Shot(float dir);

    } Option[MAX_OPTIONS];

    // 味方機の数
    int NumOptions;

    // 味方機の移動
    void MoveOptionsToShotPosition();
    void MoveOptionsToBeamPosition();

public:

    // ...(中略)...

    // 味方機が最大数かどうか
    bool HasMaxOptions() { return NumOptions==MAX_OPTIONS; }

    // 味方機の追加
    void AddOption();
};

// 自機のコンストラクタ
CMyShip::CMyShip(float x, float y, int num_options)

// ...(中略)...

{
    // 味方機の初期座標を自機と同じ座標とする
    for (int i=0; i<MAX_OPTIONS; i++) Option[i].SetXY(X, Y);
}

// 自機の移動
bool CMyShip::Move() {
```

```
    // ...(中略)...

    // 味方機の移動
    // 自機が発射しているのがショットかビームかに応じて
    // 味方機をそれぞれの隊列にするための関数を呼び分ける
    if (BeamPower<BeamMinPower) {
        MoveOptionsToShotPosition();
    } else {
        MoveOptionsToBeamPosition();
    }

    return true;
}

// 全味方機を移動させる(ショット時)
void CMyShip::MoveOptionsToShotPosition() {
    int i;

    // 画面上に出現している味方機を移動
    for (i=0; i<NumOptions; i++) {
        Option[i].MoveTo(X-5*(NumOptions-1)+10*i, Y+10);
    }

    // 未出現の味方機は自機と同じ座標にしておく
    for (; i<MAX_OPTIONS; i++) Option[i].MoveTo(X, Y);
}

// 全味方機を移動させる(ビーム時)
void CMyShip::MoveOptionsToBeamPosition() {
    int i;
    for (i=0; i<NumOptions; i++) {
        Option[i].MoveTo(X-3*(NumOptions-1)+6*i, Y+5);
    }
    for (; i<MAX_OPTIONS; i++) Option[i].MoveTo(X, Y);
}

// 味方機の移動
void CMyShip::COption::MoveTo(float x, float y) {

    // 指定された座標へのベクトルと距離を求める
    static const float SPEED=0.8f;
    float vx=x-X, vy=y-Y, l=sqrt(vx*vx+vy*vy);

    // 目標が十分に近い場合にはその座標に移動する
    if (l<SPEED) {
        X=x;
```

```
        Y=y;
    }

    // 目標が遠い場合には決められた速さでその座標に近づく
    else {
        X+=vx/l*SPEED;
        Y+=vy/l*SPEED;
    }
}

// 味方機の描画
void CMyShip::COption::Draw(float roll, float alpha) {
    Game->MeshSauce->Draw(
        X, -400, -Y, 0.6f, 0.6f, 0.6f,
        0.125f, 0, -roll, TO_ZYX, alpha, false);
}
```

## 味方機の攻撃

自機が攻撃を行ったときには、味方機も攻撃するのが一般的です。ショットとビームのように、自機の攻撃方法が複数あるときには、味方機の攻撃方法も変化させるとよいでしょう。

サンプルでは、自機が攻撃を行うと、味方機もショットを撃ちます。自機が撃ったのがショットかビームかによって、味方機はショットを撃つ角度を変化させます。ショットのときには外側へ広がるように、ビームのときには内側へ集まるように、ショットを発射します。

List 10-8は、味方機の攻撃に関するプログラムです。

**List 10-8　味方機の攻撃（MyShip.cpp）**

```
// ショットの発射
void CMyShip::Shot() {

    // 自機からショットを発射
    new CShot(X, Y, 0.70f);
    new CShot(X, Y, 0.75f);
    new CShot(X, Y, 0.80f);

    // 味方機からショットを発射
    // ショットが外側へ広がるように角度のパラメータを設定
    for (int i=0; i<NumOptions; i++) {
        Option[i].Shot(0.75f-0.03f*(NumOptions-1)+0.06f*i);
```

```
    }

    // 効果音の再生
    Game->PlaySE(Game->SEShot);
}

// ビームの発射
void CMyShip::Beam() {

    // 自機からビームを発射
    new CBeam(this, Y);

    // 味方機からショットを発射
    // ショットが内側へ集まるように角度のパラメータを設定
    for (int i=0; i<NumOptions; i++) {
        Option[i].Shot(0.75f+0.02f*(NumOptions-1)-0.04f*i);
    }

    // 効果音の再生
    Game->PlaySE(Game->SEBeam);
}

// 味方機からショットを発射
void CMyShip::COption::Shot(float dir) {
    new CShot(X, Y, dir);
}
```

## 自機爆発時のアイテム放出

自機が弾や敵などに接触して爆発したときには、味方機も失うのが一般的です。しかし、自機がやられてしまったときに、再び初期状態からパワーアップしていくのは大変なので、多くのゲームでは自機爆発時にいくつかのパワーアップアイテムを自機から放出させて、比較的楽に再パワーアップができるようにしています。

特に自機の残りをすべて失ってコンティニュー画面になったときには、大量のパワーアップアイテム、またはフルパワーアップアイテムが放出されます。これはコンティニューをうながすためです。そもそもはアーケードゲームでコインを再投入させるための工夫だったと思われますが、家庭用ゲーム機やPC用のゲームでも同様のルールを採用しているものが多く見られます。

本書では、自機を失ったときに持っていた味方機の約半分に相当する数のパワーアップ

アイテムを放出させることにしました。また、コンティニュー時にはフルパワーアップに相当する数のパワーアップアイテムを放出させます（Fig.10-6）。

アイテムの放出処理は、自機の爆発処理で行うとよいでしょう。爆発の近くにアイテムを生成することによって、自機がアイテムを放出したように見えます。

List 10-9は、自機の爆発時にパワーアップアイテムを放出するプログラムです。自機の爆発に相当するクラス（CMyShipCrash）のコンストラクタに、アイテムを放出する処理を追加します。

**Fig. 10-6** アイテムの放出

自機破壊時のアイテム放出

**List 10-9** アイテムの放出（Effect.cpp）

```
// 自機の爆発を表すクラスのコンストラクタ
CMyShipCrash::CMyShipCrash(float x, float y, int num_options)
:   CEffect(x, y), Time(0), NumOptions(num_options)
{
    // 爆発音の再生
    Game->PlaySE(Game->SEBossCrash);

    // 残機を減らす
    Game->DecNumShips();

    // 残機がある場合：
    // 現在の味方機の約半分に相当するパワーアップアイテムを放出
    if (Game->GetNumShips()>0) {
        for (int i=0, n=(NumOptions+1)/2; i<n; i++) {
            new CPowerUpItem(
                X+Game->Rand05()*10, Y+Game->Rand05()*10);
        }
```

```
    }

    // 残機がない場合：
    // フルパワーアップに十分な数のアイテムを放出
    else {
        for (int i=0, n=CMyShip::MAX_OPTIONS; i<n; i++) {
            new CPowerUpItem(
                X+Game->Rand05()*10, Y+Game->Rand05()*10);
        }
    }
}
```

## >>Chapter 10のまとめ

本章ではアイテムの出現、自動回収、得点アイテム、パワーアップアイテム、味方機、アイテムの放出などについて解説しました。パワーアップの仕組みがあると、ミスをするかしないかによって自機の強さが変わるため、同じ面でも何度も新鮮な気分で楽しむことができます。また、ミスをしないように進もうという動機も強くなります。さらに、その動機を逆手に取って、ミスをしないでいるとゲームの難易度がだんだん上がっていくようにもできます。「バトルガレッガ」(アーケード/セガサターン)などは、上がっていく難易度を適切なレベルに保つために、わざとミスして自機を失うプレイスタイルを勧めるという、独特のゲーム性を持っています。

また、得点アイテムが半自動で次々に回収される場合には、アイテムを取ったときの効果音を工夫するとよいでしょう。アイテムを連続して取ると、効果音が派手に鳴り響いて、遊んでいて快いゲームにすることが可能です。

次章ではボム、スロー、かすりといったさまざまな特殊攻撃の作り方を説明します。

# Chapter 11

# 特殊攻撃

多くのシューティングゲームでは、ショットやビームといった通常の攻撃手段の他に、ボムなどの特殊な攻撃方法が用意されています。
本章では特殊攻撃の例として、ボムとかすりの実現方法を解説します。
独特の特殊攻撃があるゲームは、プレイヤーに強い印象を与えます。
工夫を凝らして、魅力的な攻撃を作りたいところです。

Shooting Game Programming

# ボム

ボムは多くのゲームで採用されている特殊攻撃です。一般に、ボムはボタン（ボムボタン）を使って発射します。発射されたボムは、画面上の弾を消したり、敵にダメージを与えたりします。ゲームによっては、ボムの有効期間中は自機が無敵になる場合もあります。ボムを発射するタイミングに関しては、ボタンを押した瞬間にボムが出るゲームと、少し遅れてボムが出るゲームとがあります。

本章ではこういった特殊攻撃について解説します。本章のプロジェクトは「ShtGame_Special」フォルダに収録しています。実行ファイルは「ShtGame_Special¥Release¥ShtGame.exe」です。

サンプルでは、例として4種類のボムを作りました。

## ノーマルボム

画面上の弾を消し、敵にダメージを与える一般的なボムです。

## スローボム

弾や敵の動きをスローにするボムです。

## アトラクトボム

爆発の中心に画面上の弾を吸い寄せるボムです。

**Fig. 11-1** ボムの選択

### ストップボム

弾と敵の動きを完全に停止させるボムです。

※

ボムの種類はタイトル画面で選択できるようにしました（Fig. 11-1）。TYPE Aを選択すると、ショット時にはノーマルボム、ビーム時にはスローボムを発射します。TYPE Bを選択した場合には、それぞれアトラクトボムとストップボムになります。

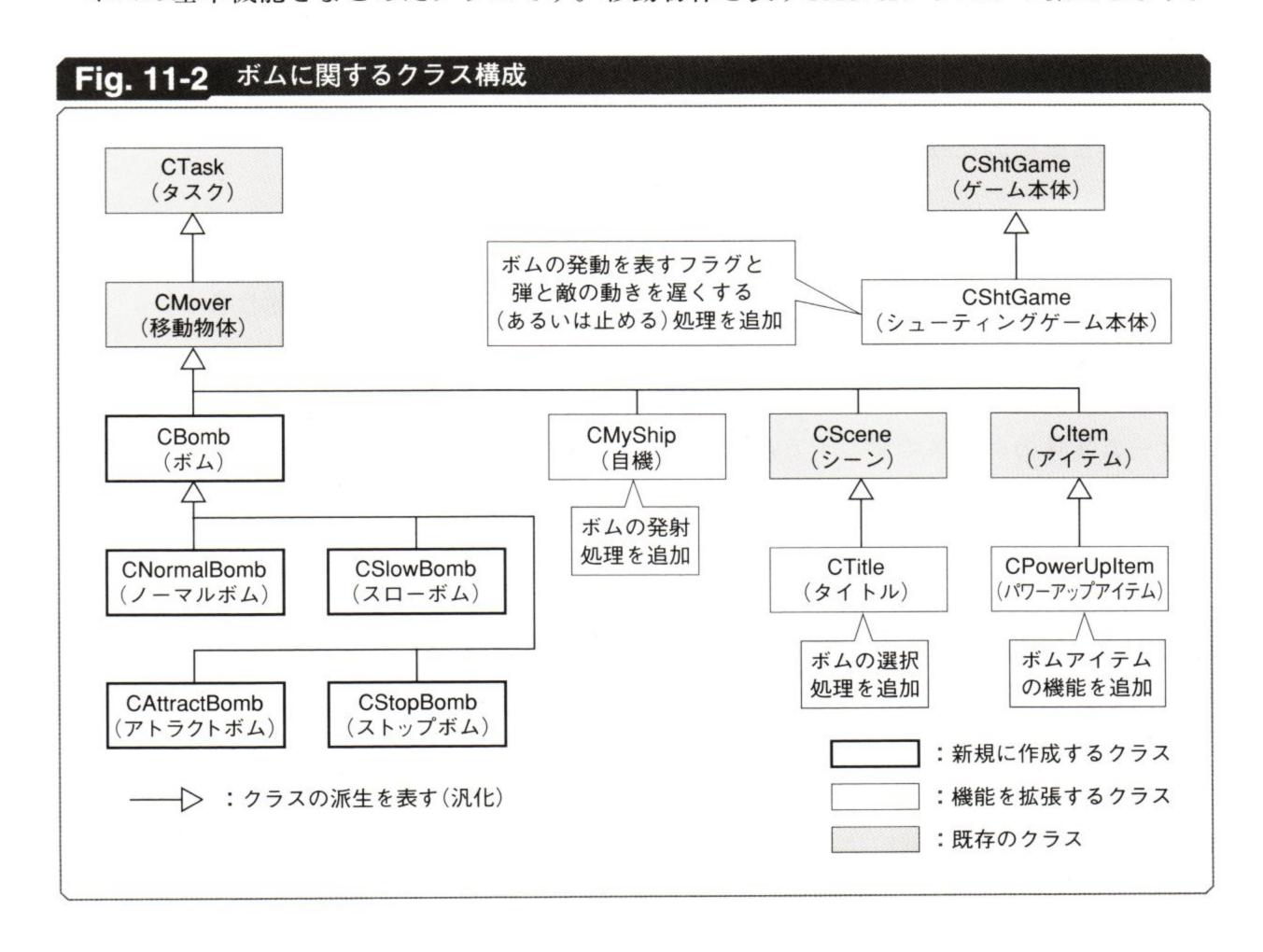
## ボムに関するクラス

Fig. 11-2はボムに関するクラス構成です。各種ボムのクラスを新規に作成する他、自機のクラスにボムを発射する処理を追加します。また、スローボムやストップボムを実現するために、ゲーム本体のクラスも拡張します。

### CBomb

ボムの基本機能をまとめたクラスです。移動物体を表すCMoverクラスから派生します。

**Fig. 11-2** ボムに関するクラス構成

### CNormalBomb

ノーマルボムです。CBombクラスから派生します。

### CSlowBomb

スローボムです。CBombクラスから派生します。

### CAttractBomb

アトラクトボムです。CBombクラスから派生します。

### CStopBomb

ストップボムです。CBombクラスから派生します。

### CMyShip

自機の基本機能をまとめたクラスです。ボムの発射処理を追加します。

### CTitle

タイトル画面のクラスです。ボムの選択処理を追加します。

### CPowerItem

パワーアップアイテムのクラスです。ボムアイテムとしての機能を追加します。

### CShtGame

ゲーム本体のクラスです。ボムが発動中かどうかを表すフラグを追加します。また、スローボムとストップボムを実現するために、弾と敵の動きを遅くする（あるいは停止する）処理も作成します。

## ボムの基本機能

種類によって効果が違いますが、ボタンを押すとボムが出て、有効期間が切れると消えるという基本的な動作は同じです。そこで、すべてのボムに共通する基本機能をまとめたクラスを作りました。そして、各種のボムを表すクラスはこの基本クラスから派生させます。

ボムに関しては、まずボムが発動中かどうかを表すフラグを用意します。このフラグは

2個以上のボムを同時に発動させないために使います。ボムを生成する際にはボム発動フラグをオンにして、逆にボムを消去する際にはオフにします。フラグがオンの間は、新たなボムを発動させないようにします。

ボムの基本クラスでは、拡大率・回転角度・アルファ値を変化させながら、ボムの3Dモデルを描画します。ボムは時間とともに回転しながら大きくなり、だんだん薄くなって、最後には消えてしまいます。

ボムが発動したら、一定時間が経過するまでボムの効果が持続します。ボムの基本クラスではタイマーの管理と描画のみを行います。弾を消したりゲームの進行をスローにするといった処理は、後述する派生クラスで行います。

List 11-1は、ボムの基本機能をまとめたクラス（CBomb）です。

**List 11-1　ボムの基本クラス（Bomb.h、Bomb.cpp）**

```
class CBomb : public CMover {
protected:

    // ボムの3Dモデル
    CMesh* Mesh;

    // タイマー、ボムの有効期間
    int Time, KillTime;

    // アルファ値
    float Alpha;

public:

    // new演算子とdelete演算子
    // タスクリストにはボムタスクリスト(BombList)を指定
    void* operator new(size_t t) {
        return operator_new(t, Game->BombList);
    }
    void operator delete(void* p) {
        operator_delete(p, Game->BombList);
    }

    // コンストラクタ、デストラクタ、移動、描画
    CBomb(CMesh* mesh, int kill_time);
    virtual ~CBomb();
    virtual bool Move();
    virtual void Draw();
};
```

```
// コンストラクタ
CBomb::CBomb(CMesh* mesh, int kill_time)
:   CMover(Game->BombList, 0, 0, BOMB_Z),
    Mesh(mesh), Time(0), KillTime(kill_time)
{
    // ボム発動フラグをオンにする
    Game->SetBombActive(true);
}

// デストラクタ
CBomb::~CBomb()
{
    // ボム発動フラグをオフにする
    Game->SetBombActive(false);
}

// 移動
// タイマーを更新し、有効期間が過ぎたら消滅する
bool CBomb::Move() {
    Time++;
    return Time<KillTime;
}

// 描画
void CBomb::Draw() {

    // タイマーから拡大率、回転角度、アルファ値を計算する
    float
        time=(float)Time/KillTime,
        scale=1+time*20,
        yaw=time,
        alpha=1-time;

    // ボムの3Dモデルを描画する
    Mesh->Draw(
        X, Z, -Y, scale, scale, scale,
        0.20f, yaw, 0, TO_YX, alpha, false);
}
```

# ボムの発射

プレイヤーがボムボタンを押したら、ボムを発射します。本書ではショット時とビーム時で発射するボムを変えることによって、複数種類のボムを使い分けられるようにしました。

市販のゲームでも、例えば「怒首領蜂」シリーズ（アーケード/PS2/セガサターン）などでは、ショット時とビーム時で違うボムが発射できます。また、自機のタイプによってボムの種類が異なるゲームもあります。

ボムを発動するときには、以下の条件が満たされているのかどうかを調べます。

- ボムボタンを押していること
- ボムがすでに発動中ではないこと
- ボムのストック数が1個以上であること

以上の条件が満たされていたら、ショット発射中かビーム発射中かによって分岐し、それぞれに対応した種類のボムを発射します。そして、ボムのストック数を1つ減らします。

List 11-2は、ボムの発射に関するプログラムです。

**List 11-2　ボムの発射（MyShip.cpp）**

```
// 移動
void CMyShip::Move() {

    // ...(中略)...

    // ボムの発射:
    // ボムが発動中ではなく、ボムボタンを押して、
    // かつボムのストック数が1以上のときだけボムを出す
    if (!Game->IsBombActive() && is->Button[1] && NumBombs>0) {
        switch (Game->GetBombType()) {

            // ボムTYPE Aの処理:
            // ショット発射中はノーマルボム
            // ビーム発射中はスローボム
            case 0:
                if (BeamPower<BeamMinPower) {
                    new CNormalBomb();
                } else {
                    new CSlowBomb();
                }
                break;
```

```
            // ボムTYPE Bの処理:
            // ショット発射中はアトラクトボム
            // ビーム発射中はストップボム
            case 1:
                if (BeamPower<BeamMinPower) {
                    new CAttractBomb();
                } else {
                    new CStopBomb();
                }
                break;
        }

        // ボムのストック数を減らす
        NumBombs--;
    }

    // ...(中略)...
}
```

## ノーマルボム

ノーマルボムは、弾を消して敵にダメージを与えるボムです。弾や敵に囲まれてしまったときの緊急回避に役立ちます（Fig. 11-3）。

ノーマルボムなどの個別のボムに相当するクラスは、ボムの基本クラス（List 11-1）から派生させます。そして、コンストラクタや移動処理を定義して、それぞれのボムに固有の処理を実装します。

**Fig. 11-3** ノーマルボム

ボムが発動したら、ボムの効果音を再生します。ボムの効果音には爆発音を使うことが多いのですが、本書では大きくなりながら消えていくボムの見た目に合わせて、「ひゅーん」という感じの音を使いました。

ノーマルボムの場合には、画面上にあるすべての弾を消し、さらにすべての敵にダメージを与えます。ボムの大きさを変化させたり、時間とともに薄くして消去したりといった処理は、ボムの基本クラスで行っています(P. 333)。

List 11-3は、ノーマルボムのプログラムです。

**List 11-3 ノーマルボム (Bomb.h、Bomb.cpp)**

```
// ノーマルボムのクラス
class CNormalBomb : public CBomb {
public:

    // コンストラクタ、移動
    CNormalBomb();
    virtual bool Move();
};

// コンストラクタ
CNormalBomb::CNormalBomb()
:   CBomb(Game->MeshYunomiBlue, 30)
{
    // 効果音の再生
    Game->PlaySE(Game->SEShotBomb);
}

// 移動
bool CNormalBomb::Move() {

    // 画面上の弾を消す
    for (CTaskIter i(Game->BulletList);
        i.HasNext(); i.Remove()) {
        ((CBullet*)i.Next())->Crash();
    }

    // 画面上の敵にダメージを与える
    for (CTaskIter i(Game->EnemyList); i.HasNext(); ) {
        ((CEnemy*)i.Next())->Vit--;
    }

    // ボムの基本クラス(CBomb)の移動処理を呼び出す
    return CBomb::Move();
}
```

# スローボム

スローボムは、弾と敵の動きを一時的に遅くします。サンプルでは、通常時の半分にスピードが落ちるようにしました。弾は消さないのですが、動きが遅くなるので、密集した弾幕も比較的簡単にすり抜けることが可能になります。後述する「かすり」と組み合わせると、スコア稼ぎにも利用できます。

「エスプガルーダ」(アーケード/PS2)にはボタンを使って弾や敵の動きをスローにする特殊攻撃があります。このスローボムは同様の攻撃をボムにしたものです。

ノーマルボムと同様に、スローボムのクラスもボムの基本クラスから派生させます。スローボムを生成したら、効果音を再生し、弾や敵を遅くするための設定を行います。スローボムを消去する際には、逆に弾や敵の動きを通常の速さに戻します。

実際に弾や敵の動きを遅くする処理は、ゲーム本体の移動処理のなかで行います。ゲーム本体の移動処理では、弾や敵といったタスクを動作させる処理を行っています。この部分で、弾や敵といったタスクを動作させる回数を間引けば、弾や敵の動きを遅くすることができます。

ここでは動きを遅くする度合いを、2や3といったパラメータで指定することにしました。指定されたパラメータが表す回数あたりに1回だけ弾や敵を動かします。

例えば、パラメータに2を指定した場合には、自機が2回動く間に弾と敵は1回しか動かないため、弾と敵の速さは半分になります。同じように、3を指定すれば速さは1/3に、4を指定すれば速さは1/4になります。

1を指定した場合には、弾と敵は通常どおりの速さで動きます。一方、0を指定した場合には弾と敵を動かさないことにしました。これはストップボムを実現するために使います(P. 342)。

List 11-4は、スローボムのプログラムです。

**List 11-4　スローボム(Bomb.h、Bomb.cpp、Main.cpp)**

```
// スローボムのクラス
class CSlowBomb : public CBomb {
public:

    // コンストラクタ、デストラクタ
    CSlowBomb();
    virtual ~CSlowBomb();
};
```

```
// コンストラクタ
CSlowBomb::CSlowBomb()
:   CBomb(Game->MeshYunomiRed, 150)
{
    // 効果音の再生
    Game->PlaySE(Game->SEBeamBomb);

    // 弾や敵の動きを半分の速さにする
    Game->SetSlowRate(2);
}

// デストラクタ
CSlowBomb::~CSlowBomb()
{
    // 弾や敵の動きを通常の速さにする
    Game->SetSlowRate(1);
}

// ゲーム本体の動作
void CShtGame::Move() {

    // ...(中略)...

    // タスクの動作
    if (!Paused) {

        // スローとストップのための処理:
        // SlowRateが2以上のときは、
        // 弾と敵をSlowRate回に1回だけ動かす。
        // SlowRateが0以下のときは、弾と敵を動かさない
        if (SlowRate>0 && Time%SlowRate==0) {
            MoveTask(BulletList);
            MoveTask(EnemyList);
        }

        // 弾と敵以外のタスクはスローとストップの影響を受けない
        MoveTask(BackList);
        MoveTask(BeamList);
        MoveTask(BombList);
        MoveTask(EffectList);
        MoveTask(ItemList);
        MoveTask(MyShipList);
        MoveTask(ShotList);
        MoveTask(TextList);

        // タイマーの更新
```

```
            Time++;
        }

        // シーンのタスクを動かす
        //(シーンのタスクは一時停止中にも動く)
        MoveTask(SceneList);

        // ...(中略)...

    }
```

## アトラクトボム

アトラクトボムは、画面上の弾をボムの中心に吸い寄せる特殊攻撃です(Fig. 11-4)。前述のノーマルボムは画面上の弾を消去しますが、アトラクトボムは中心まで弾を吸い寄せてから消去します。そのため、特に弾が大量にある場面では、吸い寄せられた弾が面白い軌跡を描きます。

アトラクトボムはノーマルボムを応用して作ることができます。ノーマルボムでは画面上のすべての弾を消しますが、アトラクトボムでは弾をボムの中心に吸い寄せる処理を行います。

そのためには、画面上のすべての弾について、ボムの中心から弾へのベクトルと距離を計算します。そして、ボムの中心付近まで近づいた弾は消去し、まだ遠くにある弾は中心に向かって吸い寄せます。

**Fig. 11-4** アトラクトボム

なお、弾に対して加える速度の向きを逆にすれば、弾を反発するボムも簡単に作ることができます。こういった特殊攻撃は、見た目やゲームバランスを考慮しつつ、実際にゲームをプレイしながら作っていくとよいでしょう。

List 11-5は、アトラクトボムのプログラムです。

**List 11-5 アトラクトボム（Bomb.h、Bomb.cpp）**

```
// アトラクトボムのクラス
class CAttractBomb : public CBomb {
public:

    // コンストラクタ、移動
    CAttractBomb();
    virtual bool Move();
};

// コンストラクタ
CAttractBomb::CAttractBomb()
:   CBomb(Game->MeshYunomiInvBlue, 150)
{
    // 効果音の再生
    Game->PlaySE(Game->SEShotBomb);
}

// 移動
bool CAttractBomb::Move() {

    // 画面上の弾をボムの中心に吸い寄せる
    static const float SPD=1.2f, DIST=1.0f;
    for (CTaskIter i(Game->BulletList); i.HasNext(); ) {
        CBullet* bullet=(CBullet*)i.Next();

        // 弾からボムの中心へのベクトルと距離の計算
        float
            vx=X-bullet->X, vy=Y-bullet->Y,
            l=sqrt(vx*vx+vy*vy);

        // ボムの中心付近まで吸い寄せた弾は消去する
        if (l<DIST) {
            bullet->Crash();
            i.Remove();
        }

        // 遠い弾は中心に向かって吸い寄せる
        else {
```

```
            bullet->X+=vx*SPD/l;
            bullet->Y+=vy*SPD/l;
        }
    }

    // ボムの基本クラス(CBomb)の移動処理を呼び出す
    return CBomb::Move();
}
```

## ストップボム

ストップボムは、画面上の弾や敵の動きを止めるボムです(Fig. 11-5)。このボムは「式神の城Ⅲ」(アーケード)に登場します。

ストップボムはスローボムを応用して実現することができます。スローボムでは弾や敵の動きを遅くするための設定を行いましたが、ストップボムでは弾や敵の動きを止めるための設定を行います。スローボムのところで解説したように、動きを遅くするパラメータに0を指定すると、弾や敵の動きが止まる仕掛けになっています(List 11-4)。

ストップボムが発動したら、効果音を再生するとともに、動きを遅くするパラメータを0にします。ストップボムが消滅するときには、パラメータを1にして、弾や敵が通常の速さで動くようにします。

List 11-6は、ストップボムのプログラムです。

**Fig. 11-5** ストップボム

**List 11-6 ストップボム (Bomb.h、Bomb.cpp)**

```
// ストップボムのクラス
class CStopBomb : public CBomb {
public:

    // コンストラクタ、デストラクタ
    CStopBomb();
    virtual ~CStopBomb();
};

// コンストラクタ
CStopBomb::CStopBomb()
:   CBomb(Game->MeshYunomiInvRed, 120)
{
    // 効果音の再生
    Game->PlaySE(Game->SEBeamBomb);

    // 弾や敵の動きを止める
    Game->SetSlowRate(0);
}

// デストラクタ
CStopBomb::~CStopBomb()
{
    // 弾や敵の動きを通常の速さに戻す
    Game->SetSlowRate(1);
}
```

## ストップボムの画面効果

ストップボムの発動時には、弾や敵の動きが止まります。このとき、ただ弾や敵を止めるだけではインパクトに欠けるので、止まっている弾や敵はワイヤーフレームで描画することによって、見た目に変化をつけてみました。

弾と敵の描画処理を拡張して、動きが止まっているときには3Dモデルをワイヤーフレームで描画します。Direct3Dでワイヤーフレーム描画を行うには、IDirect3DDevice9::SetRenderState関数を使います。D3DRS_FILLMODEにD3DFILL_WIREFRAMEを設定すると、ワイヤーフレーム描画になります。通常の塗りつぶされたポリゴン描画に戻すには、D3DFILL_SOLIDを設定します。

List 11-7は、ストップボムの画面効果に関するプログラムです。これは敵の描画処理ですが、弾の描画処理も同じ要領で行います。詳細は付録CD-ROMの「Bullet.cpp」をご覧ください。

**List 11-7　ストップボムの画面効果 (Enemy.cpp)**

```
// 敵の描画
void CEnemy::DrawMesh(
    float x, float y, float z,
    float sx, float sy, float sz,
    float tx, float ty, float tz, TURN_ORDER to,
    float a, bool aa
) {
    LPDIRECT3DDEVICE9 d=Game->GetGraphics()->GetDevice();

    // ストップボムの発動中ならばワイヤーフレームで描画する
    if (Game->GetSlowRate()==0) {

        // ワイヤーフレームで描画するための設定
        d->SetRenderState(D3DRS_FILLMODE, D3DFILL_WIREFRAME);

        // 3Dモデルをワイヤーフレームで描画
        Mesh->Draw(x, y, z, sx, sy, sz, tx, ty, tz, to, a, aa);

        // 通常のポリゴン描画に戻す
        d->SetRenderState(D3DRS_FILLMODE, D3DFILL_SOLID);
    }

    // ストップボムの発動中でなければ通常のポリゴンで描画する
    else {
        Mesh->Draw(x, y, z, sx, sy, sz, tx, ty, tz, to, a, aa);
    }
}
```

# ボムアイテム

多くのゲームでは、ゲームスタート以後にボムの所持数がまったく増えないということはなく、なんらかの手段によってボムを増やすことができます。ボムを増やす主な手段としては、スコアに応じて増やす方法と、アイテムを使う方法があります。

ここではボムを増やすために、ボムアイテムを出現させます。ボムアイテムに関するプログラムは新規に作ることもできますが、ここではChapter 10で作成したパワーアップアイテム(P. 315)を流用することにしました。

パワーアップアイテムを拡張して、ボムアイテムとしての機能を追加します。アイテムの動きはパワーアップアイテムのものをそのまま流用したので、簡単に作ることができました。

具体的には、元のパワーアップアイテムのように味方機を増やすのか、ボムアイテムとしてボムのストックを増やすのかを区別するために、フラグを追加します。そして、自機にアイテムが回収されたときには、フラグに基づいて味方機またはボムを増やします。

また、描画に関しても、フラグに基づいてアイテムの外見を変えます。味方機を増やすアイテムの場合は小さな醤油ビン、ボムを増やすアイテムの場合には小さな湯呑みの3Dモデルを描画します（Fig. 11-6）。

List 11-8は、ボムアイテムのプログラムです。

**Fig. 11-6** ボムアイテム

**List 11-8** ボムアイテム（Item.h、Item.cpp）

```
// パワーアップアイテム
class CPowerUpItem : public CItem {
protected:

    // 速度
    float VX, VY;

    // 味方機を増やすかボムを増やすかのフラグ
    bool OptionOrBomb;

public:

    // コンストラクタ
    // 味方機を増やすかボムを増やすかは引数で指定する
    CPowerUpItem(float x, float y, bool option_or_bomb);

    // 移動、描画
```

```
    virtual bool Move();
    virtual void Draw();
};

// 移動
bool CPowerUpItem::Move() {

    // ...(中略)...

    // 自機に接触したとき
    if (Hit(myship)) {

        // 効果音の再生
        Game->PlaySE(Game->SEPowerItem);

        // フラグに応じて味方機またはボムを増やす
        OptionOrBomb?myship->AddOption():myship->AddBomb();

        // 自分(アイテム)の消去
        return false;
    }

    // ...(中略)...

}

// 描画
// フラグに応じてアイテムの外見を変える
void CPowerUpItem::Draw() {
    OptionOrBomb?
        Game->MeshSauce->Draw(
            X, Z, -Y, 0.6f, 0.6f, 0.6f,
            0.125f, Yaw, 0, TO_ZYX, 1, false):
        Game->MeshYunomiSmall->Draw(
            X, Z, -Y, 2, 2, 2,
            0.125f, Yaw, 0, TO_ZYX, 1, false);
}
```

## かすり

かすりとは、自機を弾や敵にかすらせることによって、スコアを得たり自機をパワーアップさせたりするルールです。このルールは「RAIDEN FIGHTERS」(アーケード)、「レイ

ディアントシルバーガン」(アーケード/セガサターン)、「サイヴァリア」(アーケード/ドリームキャスト/PS2/Xbox)、「式神の城」(アーケード/ドリームキャスト/GC/PS2/Xbox) などに採用されています。

かすりはプレイヤーに対して、ただ安全に弾を避けるのではなく、あえて弾幕のなかに突っ込んでいくプレイを要求します。これはゲームの緊張感を増すとともに、プレイヤーの技量に応じてプレイスタイルを変えることを可能にします。

こういったかすりを実現する方法は次のとおりです。自機の当たり判定の外側に、かすり用の当たり判定を作ります (Fig. 11-7)。そして、かすり用の当たり判定に弾や敵が接触したら、かすったと判断して、ボーナス点などを加算します。

ゲームによっては、同じ弾や敵にかすれる回数や間隔を制限している場合もあります。本書では弾のみにかすれるとし、また1つの弾には1度だけかすれることにしました。

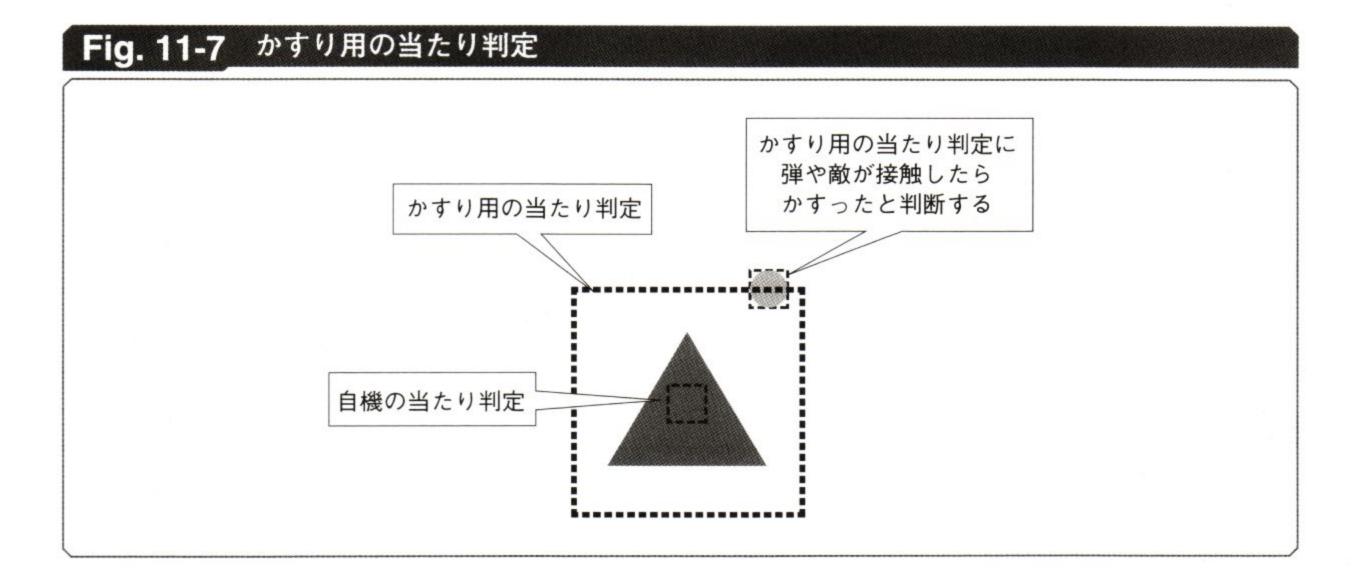
**Fig. 11-7 かすり用の当たり判定**

## かすりに関するクラス

Fig. 11-8はかすりを実現するためのクラス構成です。通常時の自機を表すクラスと弾のクラスを拡張します。また、かすりのエフェクトを表すクラスを新規に作成します。

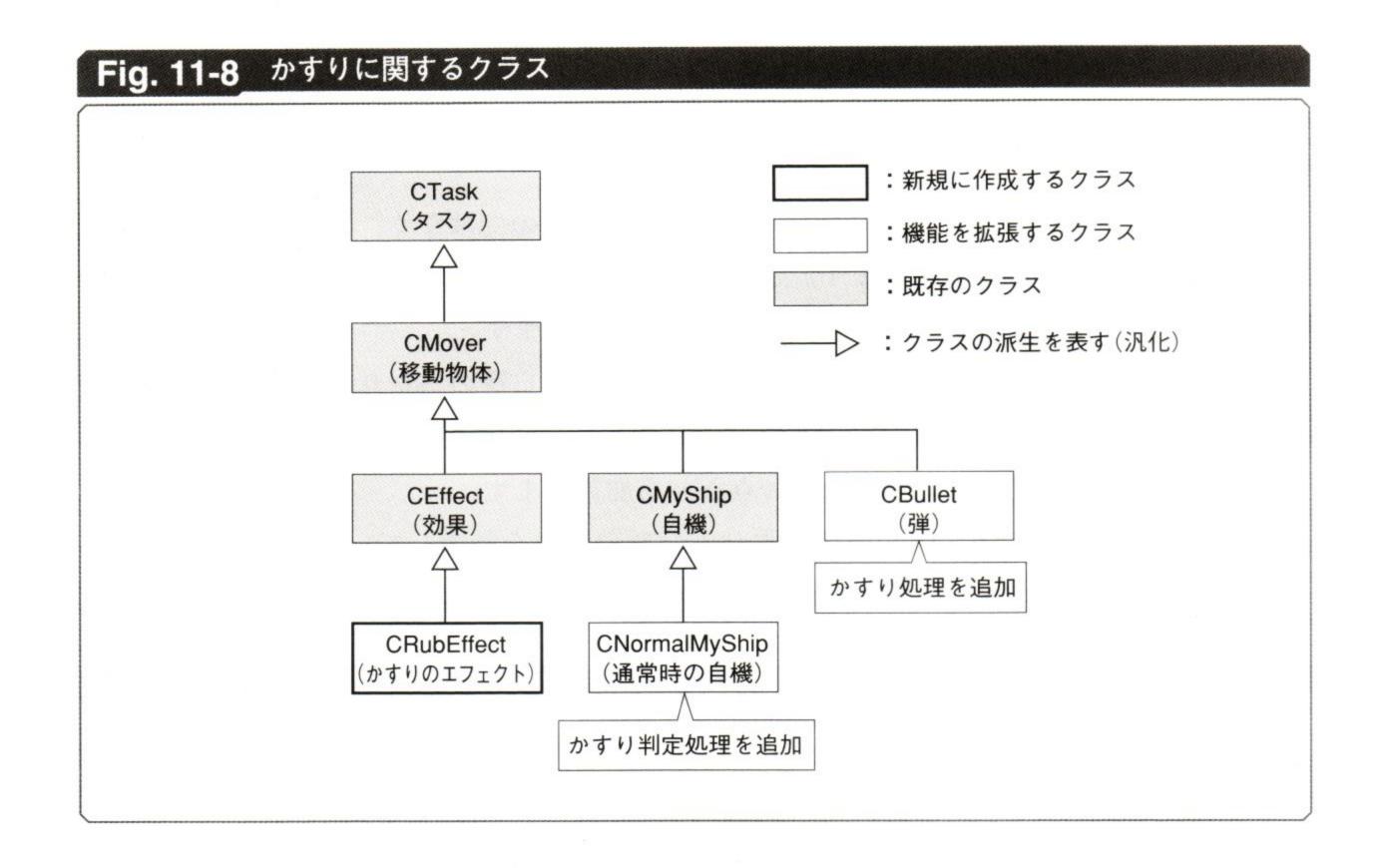
### CRubEffect

自機が弾にかすったときに表示するエフェクトのクラスです。CEffectクラス (P. 164) から派生します。

### CNormalMyShip

通常時の自機です。弾にかすったかどうかを判定する処理を追加します。

**Fig. 11-8** かすりに関するクラス

### CBullet

弾の基本機能をまとめたクラスです。自機にかすられたときの処理を追加します。

## かすりの実現方法

かすりの判定処理は、自機の移動処理に追加します。そこでは敵や弾との当たり判定処理を行いますが、続いてかすり判定処理を行うようにします。前者には自機の中心にある小さな当たり判定を使い、後者には自機の周囲を囲む大きな当たり判定を使います。

一方で、かすられた弾にも処理を追加する必要があります。かすられたかどうかを表すフラグを用意して、自機にかすられたらフラグをセットするようにします。このフラグを使って、初めてかすられたときだけエフェクトの生成や効果音の再生、スコアの加算を行います。

List 11-9は、かすりを実現するプログラムです。なお、弾にかすったら、その弾と同じ位置にかすりのエフェクトを表示します。このエフェクトについては次のList 11-10で説明します。

**List 11-9　かすりのプログラム（MyShip.cpp、Bullet.cpp）**

```
// 通常時の自機の移動
bool CNormalMyShip::Move() {

    // ...(中略)...

    // 当たり判定処理とかすり判定処理(自機と弾)
    for (CTaskIter i(Game->BulletList); i.HasNext(); ) {
        CBullet* bullet=(CBullet*)i.Next();

        // 当たり判定処理
        if (Hit(bullet)) {
            Vit--;
        } else

        // かすり判定処理
        // (弾に接触しなかった場合だけ、かすりを判定する)
        if (Hit(bullet, -5, -5, 5, 5)) {
            bullet->Rub();
        }
    }

    // ...(中略)...

}

// 弾のかすり処理
void CBullet::Rub() {

    // 初めてかすったときだけ以下の処理を行う
    if (!Rubbed) {

        // かすりのエフェクトを生成する
        new CRubEffect(X, Y);

        // かすったことを記録する
        Rubbed=true;

        // 効果音の再生
        Game->PlaySE(Game->SERub);

        // スコアの加算
        Game->AddScore(50);
    }
}
```

## かすりのエフェクト

弾や敵にかすったときには、かすりに成功したことがプレイヤーにわかるように、なんらかの演出が必要です。例えば、かすりの成功時にエフェクトを表示すれば、プレイヤーはかすりを狙うのが楽しくなるでしょう（Fig. 11-9）。

かすりのエフェクトは、爆発のエフェクトなどと似た方法で実現することができます。ここでは、星形の3Dモデルが回転しながらしだいに大きくなり、だんだん薄くなって消えるようなエフェクトにしました。経過時間に応じて拡大率・回転角度・アルファ値などを変化させながら3Dモデルを描画すれば、このようなエフェクトが表示できます。エフェクトは一定時間が経過したら消去します。

List 11-10は、かすりのエフェクトに関するプログラムです。

**Fig. 11-9** かすりのエフェクト

**List 11-10** かすりのエフェクト（Effect.h、Effect.cpp）

```
// かすりエフェクト
class CRubEffect : public CEffect {

    // タイマー
    int Time;

public:

    // コンストラクタ、移動、描画
    CRubEffect(float x, float y);
    virtual bool Move();
    virtual void Draw();
```

```
};

// コンストラクタ
CRubEffect::CRubEffect(float x, float y)
:   CEffect(x, y), Time(0)
{}

// 移動
// タイマーを更新し、約1/3秒後に消滅する
bool CRubEffect::Move() {
    Time++;
    return Time<=20;
}

// 描画
void CRubEffect::Draw() {

    // タイマーから拡大率、回転角度、アルファ値を計算する
    float
        time=(float)Time/20,
        scale=time/2,
        yaw=time/2,
        alpha=2-time;

    // かすりの3Dモデルを描画する
    Game->MeshRubEffect->Draw(
        X, Z, -Y, scale, scale, scale,
        0, yaw, 0, TO_Y, 1, false);
}
```

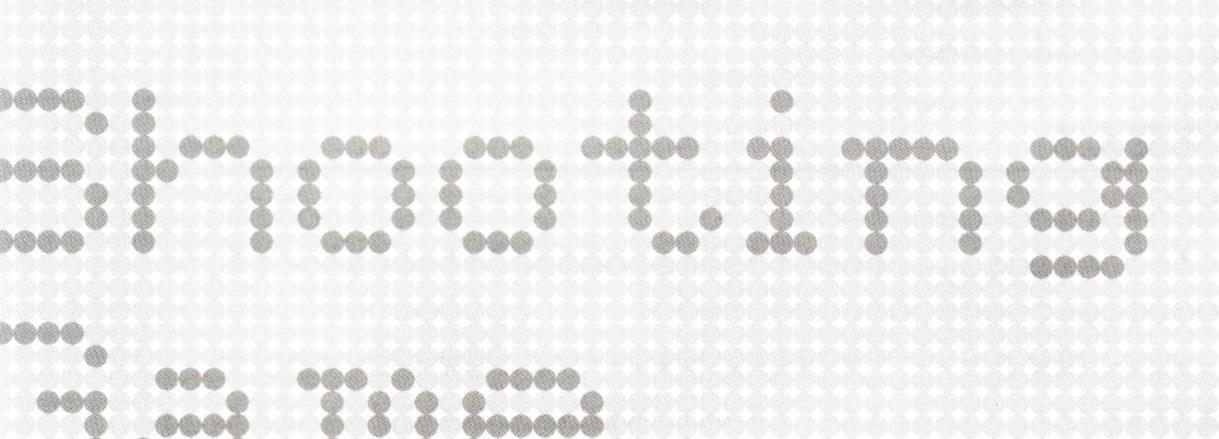
## >>Chapter 11のまとめ

本章ではボムやかすりといった特殊攻撃の実現方法を解説しました。独特の特殊攻撃を持っているゲームは魅力的です。ただし、好きなゲームから特殊攻撃のアイディアをあれこれ取り入れてゲームを作ると、盛り込みすぎてかえってバランスの悪いゲームになってしまうことがあります。楽しくプレイできるよう、バランスを考えながら特殊攻撃のデザインをするとよいでしょう。さらにオリジナリティのある特殊攻撃ならば最高です。

最後に、必要に応じて「ゲームのブラッシュアップ」を行えば、いよいよ完成です。付録CD-ROMに収録した「ChapterA.pdf」では、リプレイ機能、難易度の調整、データのアーカイブ化について解説しています。

# Chapter 12

# オリジナルゲーム制作ガイド

本書ではサンプルの制作を通じて、シューティングゲームのプログラミングについて解説してきました。本書で紹介した手法やソースコードを利用すれば、オリジナルのシューティングゲームを作ることは難しくないでしょう。とはいえ、ゼロからゲームを作るとなると、いろいろな作業が必要になります。プログラムだけではなく、グラフィックやサウンドのデータも用意しなければなりません。
そこで本章では、本書のサンプルを土台にしてオリジナルゲームを制作するためのヒントを紹介します。サンプルを簡単に改造するためのポイントやアイディアを解説するので、気軽にオリジナルのシューティングゲーム制作に挑戦してみてください。

# データを入れ替える

サンプルのグラフィックやサウンドといったデータを入れ替えるだけでも、まったく違ったゲームのようにアレンジすることができます。簡単に入れ替えられるデータとしては、次のようなものがあります。

・グラフィック
・サウンド
・スクリプト

なお、本書収録のプログラムやデータの利用規程については、本書392ページおよび付録CD-ROM収録のreadme.txtをご参照ください。

## グラフィック

グラフィックはいちばん効果的な改造ポイントです。グラフィックデータを入れ替えると、ゲームの見た目は大きく変わり、まったく別のゲームのようになります。

本書のサンプルでは、自機や敵のキャラクターを3Dグラフィックを使って表示します。これらのキャラクターの3Dモデルを入れ替えれば、ゲームの見た目を変えることができます。のちほど、実際にグラフィックを入れ替えた例を紹介します。

本書のサンプルでは、データファイルの種類別にフォルダを分けています。例えば、自機の3Dモデルは「Model¥sauce.x」です。3DツールでDirectX形式（.x）の3Dモデルを作り、「Model¥sauce.x」に上書きすれば、自機の3Dモデルを変更することができます。この場合、プログラムを変更する必要はありません。

## サウンドを変える

サウンドも手軽に入れ替えることができます。ただ、サウンドだけを入れ替えるのではなく、まずはグラフィックを変更して、そのグラフィックの雰囲気に合わせてサウンドも変更するのが効果的でしょう。

本書のサンプルの場合、効果音のデータは「Sound」フォルダ以下に格納しています。ファイルはWave形式（.wav）です。Wave形式のファイルは、Windowsに標準で付属するサウンドレコーダをはじめとして、多くのサウンドツールで作成することができます。

一方、音楽のデータは「Music」フォルダ以下に格納しています。ファイルはWave形式（.wav）の他、MIDI形式（.mid）、WMA形式（.wma）、MP3形式（.mp3）などが使えます。

### スクリプトを変える

グラフィックやサウンドを入れ替えただけでは、ゲームの見た目は変わっても、ゲームの内容はまったく変わりません。ゲームの内容を手軽に変える方法としては、スクリプトファイルに書かれた敵の出現シーケンスを変更する方法があります。

サンプルのスクリプトファイルは、「script.txt」というテキストファイルです。このファイルには、敵の出現位置や出現タイミングの他、BGMの再生や停止に関する指示も書くことができます。詳しい書き方はChapter 8で解説しました（P. 269）。

敵の出現シーケンスを変更すれば、たとえプログラムに手を入れなくても、ゲームの内容を元とはかなり違ったものにすることができます。

## グラフィックを入れ替える例

データを入れ替える例として、実際にサンプルのグラフィックを入れ替えてみました。元のグラフィックは寿司をモチーフにしていますが、ここでは菓子をモチーフにしたバージョンを用意しました（Fig. 12-1、2）。Fig. 12-1、2の左右の画面はそれぞれ同じシーンですが、キャラクターの3Dモデルが違うので、かなり違った雰囲気になっています。

ここでは、自機と敵の3Dモデルを入れ替えました。例えば、寿司バージョンの赤身や玉子は、菓子バージョンではショートケーキやチーズケーキに変わっています。自機は醤油ビンからクマの人形に変わりました。

また、スコア領域に表示する残機の2D画像を変えたり、スコア領域の背景色を変えた

**Fig. 12-1** グラフィックの入れ替え①

寿司バージョンの実行例①

菓子バージョンの実行例②

**Fig. 12-2** グラフィックの入れ替え②

寿司バージョンの実行例②

菓子バージョンの実行例②

りもしました。背景も寿司バージョンでは魚の漢字だったものを、菓子バージョンではパステルカラーのチェック模様に変えています。

このようにグラフィックを入れ替えると、ゲームの雰囲気は大きく変わります。グラフィックを変更するには、元のファイルと同じファイル名で新しいデータを上書きするだけです。プログラムを変更する必要はまったくありません。

本章のサンプルでは、寿司バージョンと菓子バージョンのデータを両方とも読み込んでいます。そして、タイトル画面の「SKIN」という項目で、寿司または菓子を切り替えられるようにしています。このように、複数のグラフィックを用意しておき、プレイヤーが好みに応じて見た目を選べるようにしても面白いでしょう。

なお、本章のプロジェクトは「ShtGame_Arrange」フォルダに収録しました。実行ファイルは「ShtGame_Arrange¥Release¥ShtGame.exe」です。

# プログラムを改造する

データを入れ替えるだけではなく、プログラムにも手を加えることができれば、ゲームの内容をより大きく変化させることができます。ゼロから全部のプログラムを作るのは大変でも、例えば弾や敵のプログラムだけを書くのはそれほど難しくはありません。こういった気軽な改造から始めて、ゆくゆくは完全にオリジナルのゲームを作り上げていけばよいでしょう。

プログラムには、次のような改造ポイントがあります。

### 弾や弾幕のアレンジ

本書のChapter 5では、方向弾・狙い撃ち弾・誘導弾などについて解説し、それらを組み合わせて弾幕を作る方法も紹介しました。こういった弾のプログラムを書くのは難しくはありません。本書で紹介した弾を応用すれば、さまざまな種類の弾を作ることができます。また、これらの弾を組み合わせて、いろいろな弾幕を作ることもできます。

### 敵の挙動のアレンジ

弾と同様に、敵のプログラムも比較的簡単に書くことができます。本書のChapter 6ではザコの作り方を解説し、Chapter 8では少し大型の敵を作りました。また、Chapter 9ではボスの作り方も紹介しました。

最初は既存の弾や敵を応用して、何かオリジナルの弾か敵を1つか2つ作ってみるとよいでしょう。そして、だんだんと独自の弾や敵を増やしていけば、いつの間にかサンプルとはまったく違うオリジナルのゲームになっているはずです。

## 特殊攻撃を追加する①「ソード」

本書のChapter 11では、ボムやかすりといった特殊攻撃について解説しました。シューティングゲームを個性的に見せる要素はいろいろとありますが、そのなかでも特殊攻撃は重要な要素です。特殊攻撃が魅力的なゲームは、プレイヤーに強い印象を与えます。場合によっては、特殊攻撃がゲームの中心的な要素となり、ゲーム性を決定することさえあります。

サンプルを元にオリジナルのゲームを制作するときには、特徴のある特殊攻撃を追加することから始めるのも1つの方法です。その特殊攻撃を中心にゲームを組み立てることによって、元とはまったく違ったゲームに仕上げることも可能でしょう。

ここでは特殊攻撃の一種である「ソード（剣）」の制作例を紹介します。そして、ショットやビームのかわりにソードだけを使ったゲームにアレンジすることによって、元とはまったく違った内容のゲームにします。

本章のサンプルでは、タイトル画面で「START（SWORD）」を選択すると、ソードを使ったゲームを遊ぶことができます。このモードでは、ショット・ビーム・ボムはいっさい使わず、ソードだけを使って遊びます。

ソードは弾を斬って消すことができます。また、敵を斬って破壊することもできます

(Fig. 12-3)。近接攻撃なので弾や敵に接近しなければならず、プレイには緊張感が伴います。その一方で、ショットやビームに比べてソードの攻撃力はかなり高いので、弾や敵を次々と斬りながら爽快に進むことができるでしょう。

ソードの動きは「レイディアントシルバーガン」(アーケード/セガサターン) を参考にしました。「斑鳩」(アーケード/ドリームキャスト/GC) の前作に相当するこの作品は、非常に多くのユニークな仕掛けが組み込まれたゲームで、コアなシューティングゲーマーを中心に熱烈なファンを獲得しています。

このソードは動きに特徴があります。ここで制作するソードは、次のような動きをします。

キーボードの「Z」キーまたはジョイスティックのボタン「0」を押すとソードが出ます。ボタンを押しっぱなしにしている間はソードが出たままになります。ボタンを離すとソードは消えます。

ボタンを押すと、ソードは自機の正面に出ます。ボタンを押しっぱなしにすると、ソードは右回りに回転し、再び自機の正面にくるまで回転します (Fig. 12-4)。

1回転した後のソードは、自機が進む向きと逆の向きにくるように回転します。自機を動かすと、ソードを振り回すことができます。その際には、ソードの残像が帯状の美しい軌跡を描きます (Fig. 12-5)。

ソードの軌跡はアルファブレンディングによる加算合成を使って描画しています。そのため、光の帯のような独特の効果が得られています。

ソードの当たり判定処理は、ショットやビームとは違った独特の方法で行います。ソードの角度によって当たり判定が変わるので、ぜひ一度実際にサンプルを動かして試してみてください。

**Fig. 12-3** ソード

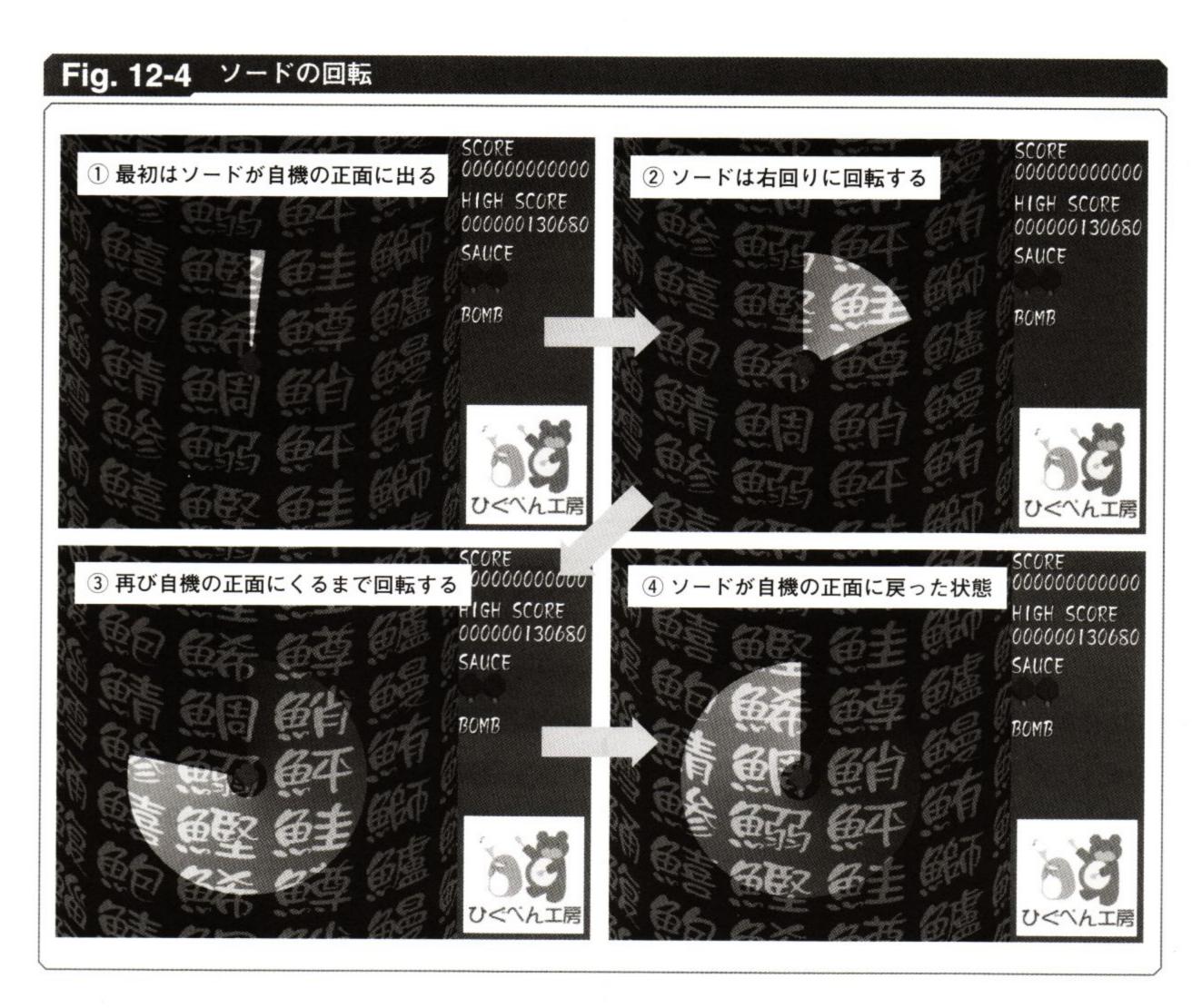
**Fig. 12-4** ソードの回転

① 最初はソードが自機の正面に出る

② ソードは右回りに回転する

③ 再び自機の正面にくるまで回転する

④ ソードが自機の正面に戻った状態

**Fig. 12-5** 回転後のソードの動き

## ソードの回転

ソードの実現するうえでの最初のポイントは、自機の移動に応じてソードを回転させる方法です。ソードは自機の移動方向とは反対の方向にきます。ただし、一瞬でこの方向にくるのではなく、現在のソードの位置からなめらかに回転して角度を変えます。

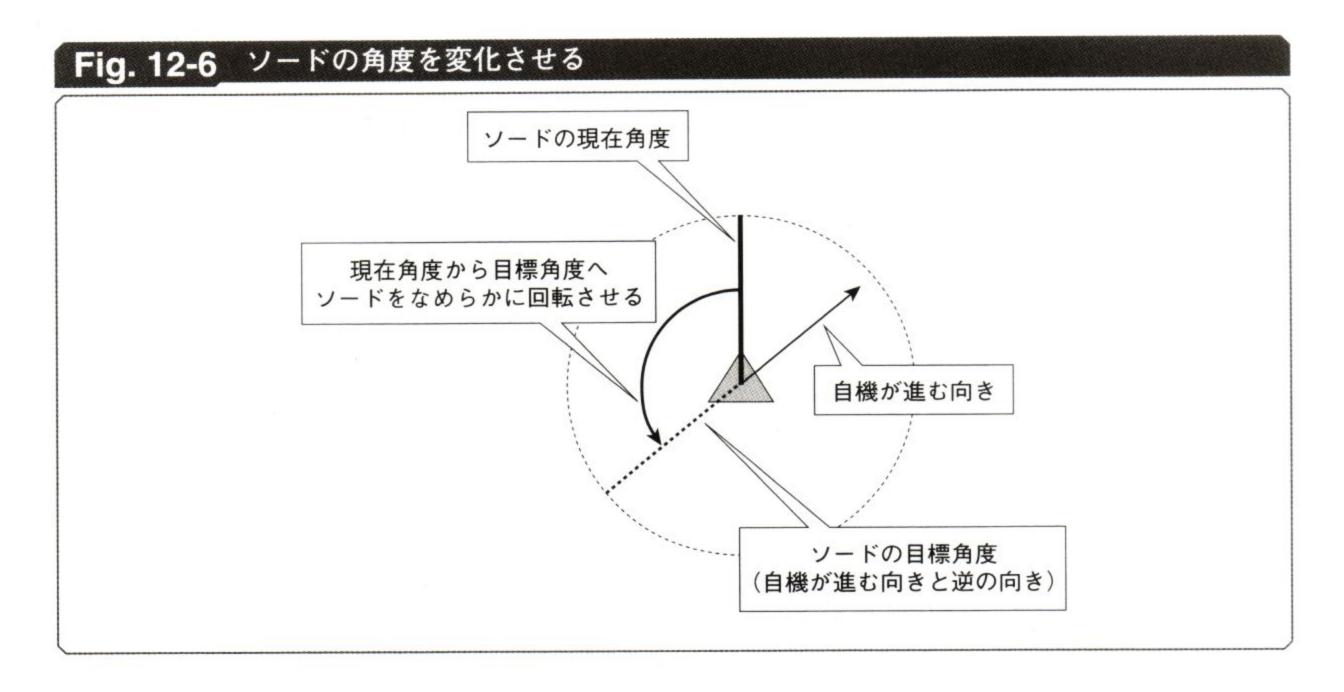
現在のソードの角度を現在角度と呼び、目指す角度を目標角度と呼ぶことにしましょう。目標角度は自機が進む向きと逆の向きです。ソードは現在角度から目標角度まで、なめらかに回転します(Fig. 12-6)。

ソードを回転させるときには、回転の向きが重要です。右回りか左回りのうち、目標角度に対して近い向きに回転させる必要があります(Fig. 12-7)。

ここでは、角度を360度あたり1.0で表すことにして説明を進めます。90度は0.25、180度は0.5です。1.0は0.0と同じで、0度(または360度)に相当します。同様に、-1.0や2.0も0度を表すとします(Fig. 12-8)。

前述のように、ソードの目標角度は自機が進む向きに応じて決めます。例えば、自機が左に移動したら、目標角度は右方向を表す0.25にします。自機が右上に移動したら、目標角度は左下を表す0.625にします。

例えば、現在角度が0.0で、目標角度が0.375の場合を考えましょう(Fig. 12-9)。この場合は、ソードを右に回転させるのが適切です。ソードを回転させるには、次のような処理を行います。

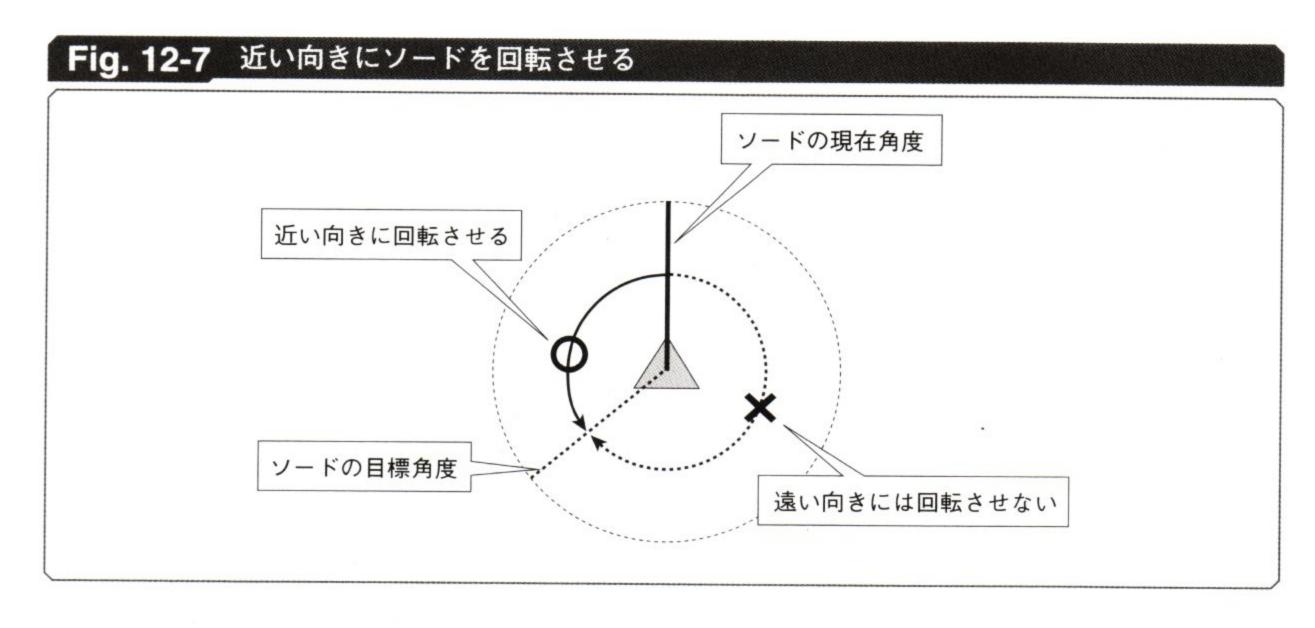
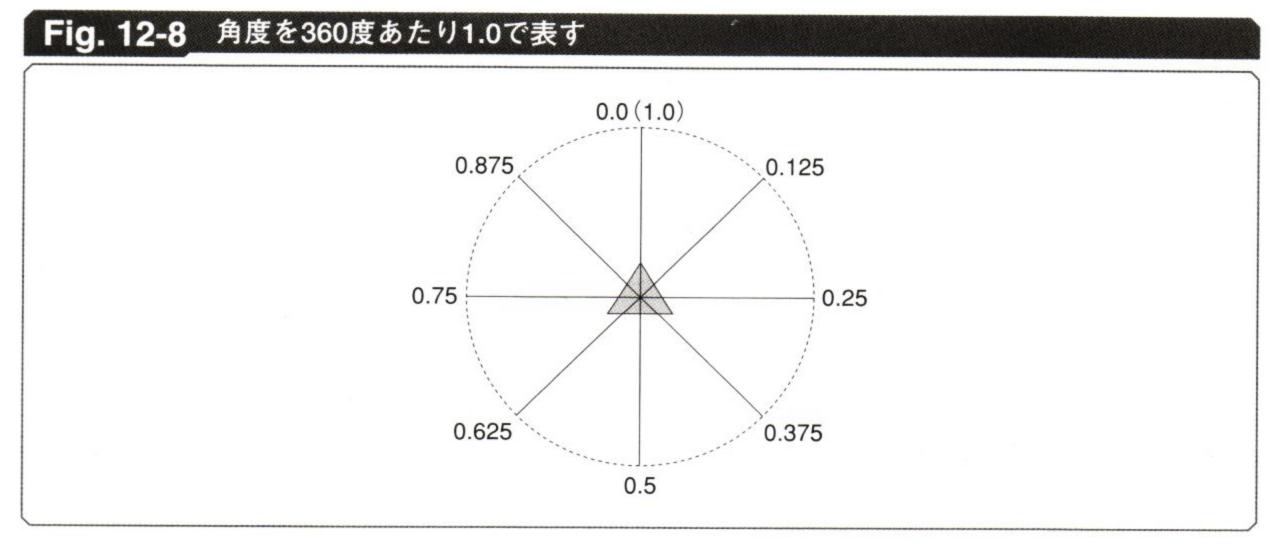
**Fig. 12-6** ソードの角度を変化させる

①目標角度から現在角度を減算して、差分を求めます。ここでは「0.375-0.0」なので、差分は0.375になります。

②差分が0.5未満の場合には右回転、0.5以上の場合には左回転にします。ここでは「0.375<0.5」なので、右回転です。

③右回転の場合には、現在角度に差分の10%程度を加算します。加算する割合を大きくすると、ソードの回転が速くなります。ここでは「0.0+0.375＊0.1=0.0375」が新しい現在角度になります。

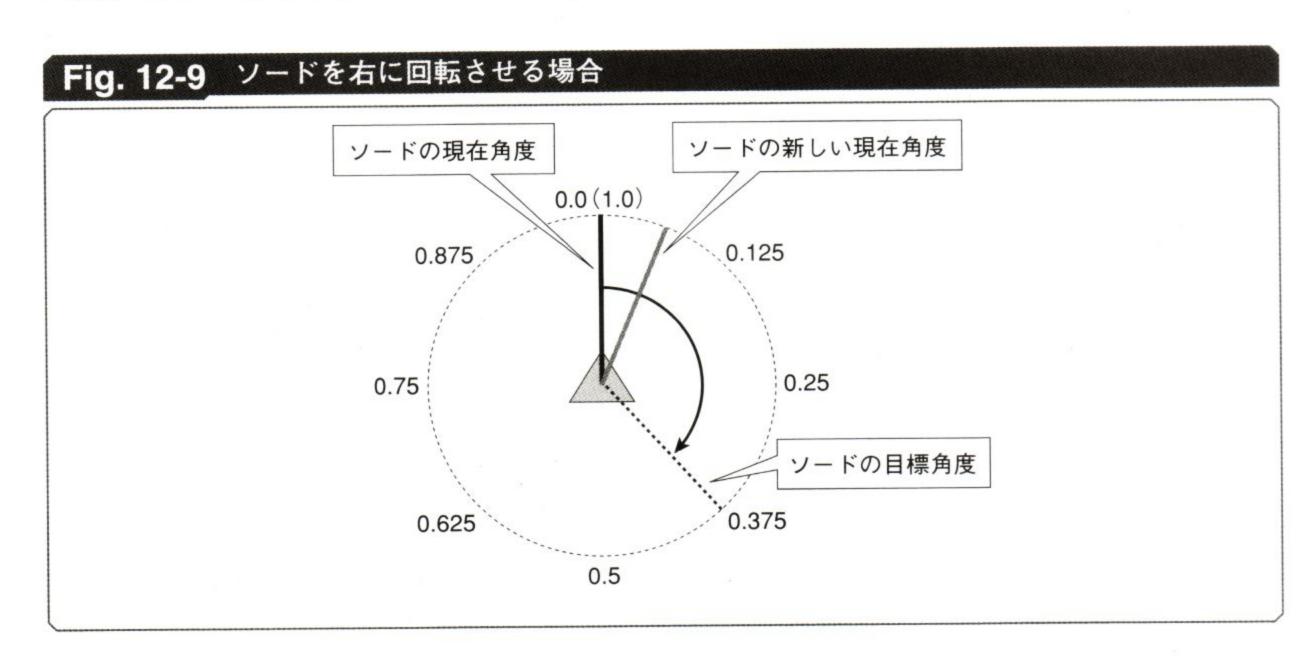
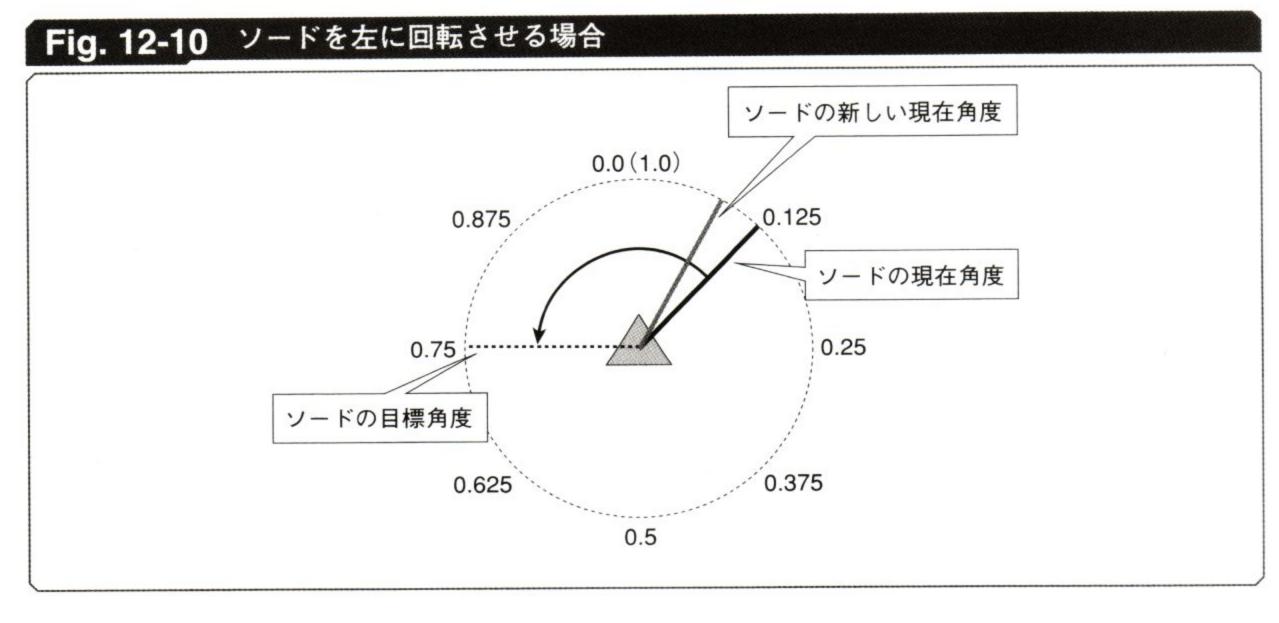
次に、左に回転させる場合も考えましょう。例えば、現在角度が0.125で、目標角度が0.75とします（Fig. 12-10）。この場合は次のような処理を行います。

**Fig. 12-7** 近い向きにソードを回転させる

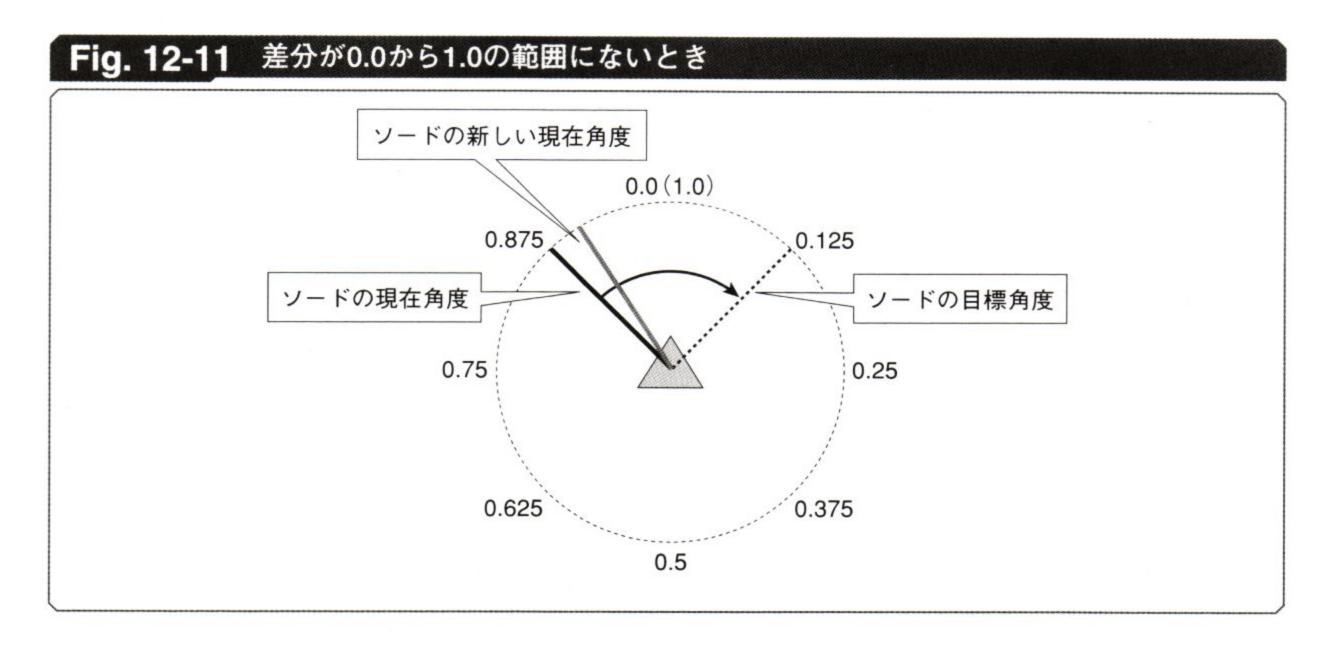
**Fig. 12-8** 角度を360度あたり1.0で表す

①目標角度から現在角度を減算して、差分を求めます。ここでは「0.75-0.125」なので、差分は0.625になります。
②差分が0.5未満の場合には右回転、0.5以上の場合には左回転にします。ここでは「0.625>=0.5」なので、左回転です。
③左回転の場合には、現在角度に（差分-1）の10%程度を加算します。ここでは「0.125+（0.625-1）＊0.1=0.125-0.0375=0.0875」が新しい現在角度になります。

ここまでの方法でソードを左右に回転させることができます。ただ、差分が0.0から1.0の範囲にないときには、少し特別な処理が必要です。例えば、現在角度が0.875で、目標角度が0.125の場合を考えてみます（Fig. 12-11）。この場合は右回転が適切です。

**Fig. 12-9 ソードを右に回転させる場合**

**Fig. 12-10 ソードを左に回転させる場合**

①目標角度から現在角度を減算して、差分を求めます。ここでは「0.125-0.875」なので、差分は-0.75になります。このように差分が0.0から1.0の範囲にないときには、差分に1.0を加算または減算することにより、0.0から1.0の範囲に修正します。ここでは-0.75に1.0を加算して、差分を0.25とします。
②差分が0.5未満の場合には右回転、0.5以上の場合には左回転にします。ここでは「0.25<0.5」なので、右回転です。
③右回転の場合には、現在角度に差分の10%程度を加算します。ここでは「0.875+0.25＊0.1=0.9」が新しい現在角度になります。

差分を0.0から1.0の範囲に収めるには、floor関数（またはfloorf関数）を使うのが便利です。floor関数は、引数で指定された実数以下で最大の整数を返します。差分からfloor（差分）を減算すれば、差分から整数部分を取り除き、小数部分だけを取り出すことによって、値を0.0から1.0の範囲にすることができます（List 12-1、P. 367）。

**Fig. 12-11** 差分が0.0から1.0の範囲にないとき

## 弾や敵との当たり判定処理

ソードは回転するので、特別な当たり判定処理が必要です。ここではソードの当たり判定を扇状にしました（Fig. 12-12）。この扇状の領域内に弾や敵の中心座標が入ったら、接触したと見なして、弾を消したり敵にダメージを与えたりします。

具体的には、次のような方法で判定処理を行います（Fig. 12-13）。図には敵の場合を示しましたが、弾の場合も同様です。すべての弾と敵に関して、この方法で当たり判定処理を行います。

①自機から目標（弾または敵）の中心までの距離を計算します。距離がソードの長さ以下ならば②に進みます。

②自機から目標の中心までの方向ベクトルを求めます。この方向ベクトルとソードの方向ベクトルの内積を求めて、内積が一定値より大きければ接触したと判定します。

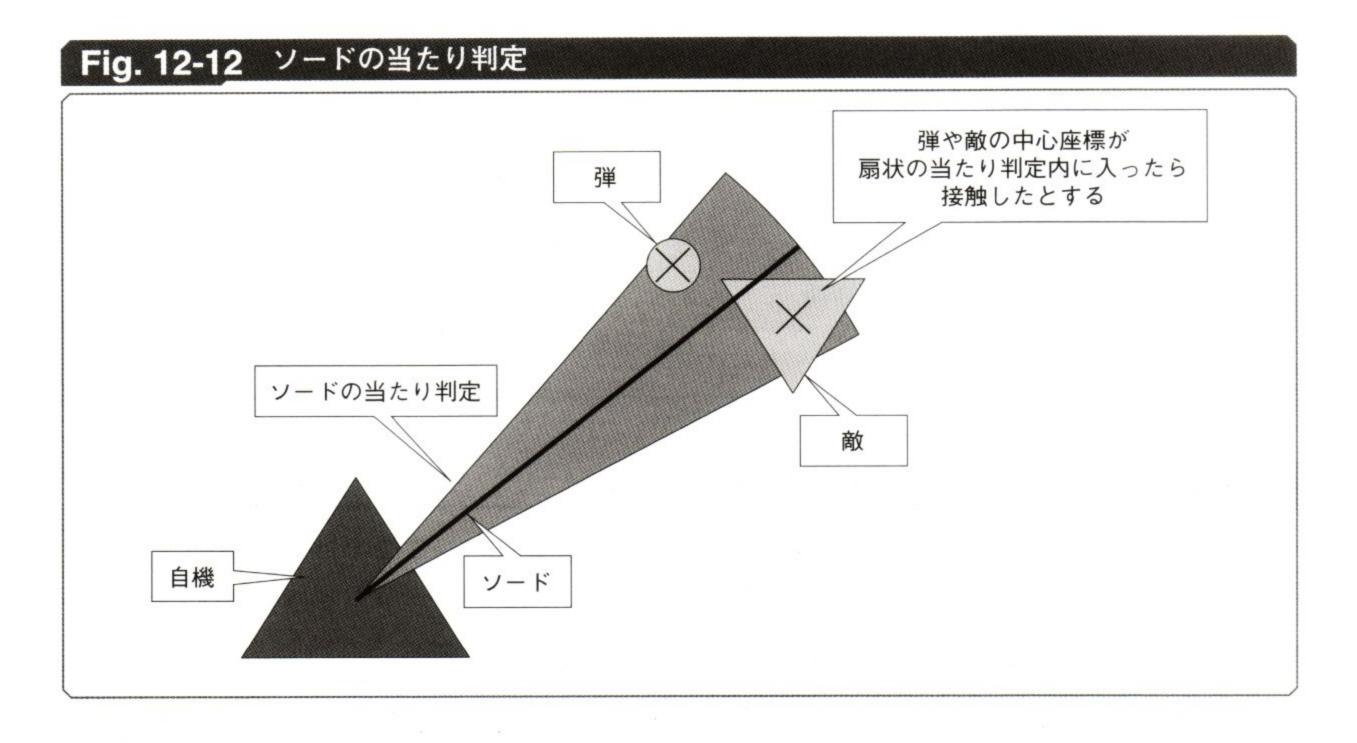
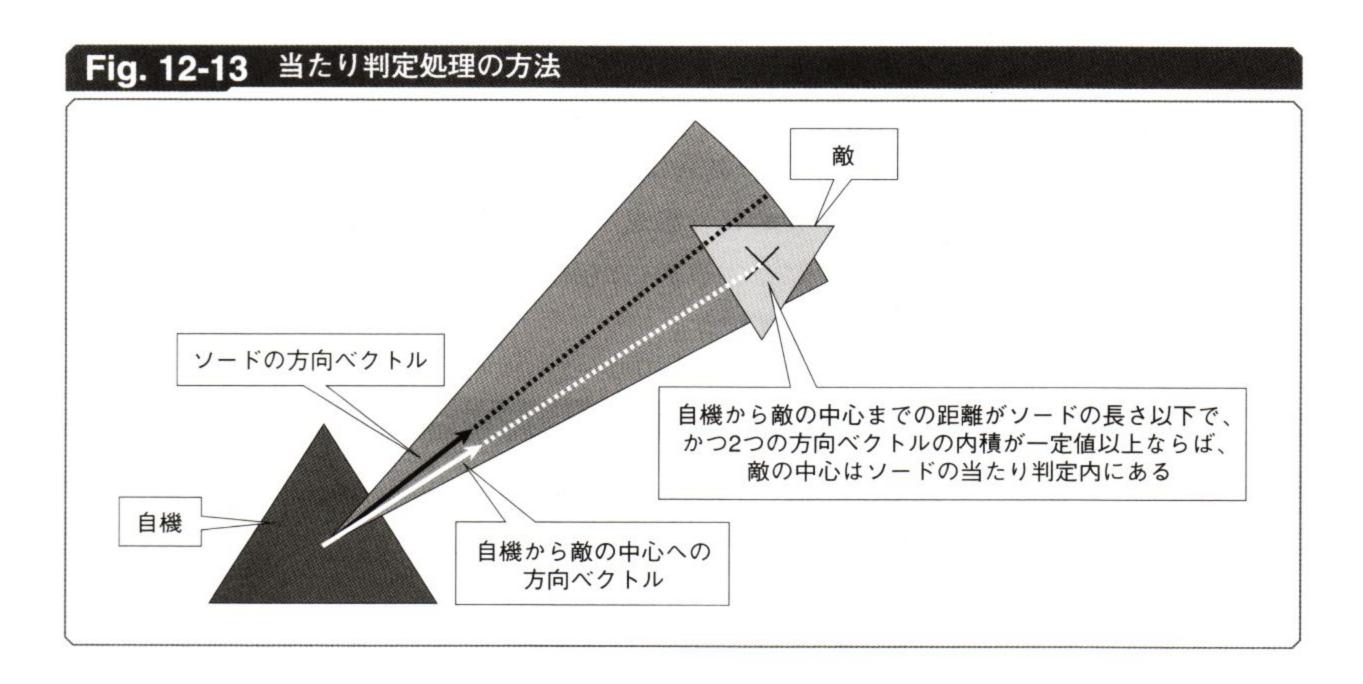
方向ベクトルの長さを固定した場合には、2つの方向ベクトルの角度が近いほど、内積の値は大きくなります。そこで、一定値よりも内積が大きいかどうかを調べれば、扇状の領域内に目標の中心が入ったかどうかを判定することができます。

ベクトルの長さが1の場合、内積の最大値は1です。当たり判定を内角が小さな扇状の領域にするには、内積と比較する値を最大値に近い値、例えば0.97などにします。

当たり判定処理の結果、ソードが弾に接触したら弾を消し、敵に接触したらダメージを与えます。ソードはショットに比べて敵に接近しなければならないぶん、攻撃力は大きくするとよいでしょう。敵に接近するリスクが大きい反面、当たれば一撃で敵を破壊できるとなれば、スリリングなゲームになります。

弾や敵を斬ったときには、斬ったことがわかるように効果音を鳴らすとよいでしょう。本書のサンプルでは、効果音に加えて、液体（醤油）が飛び散るエフェクトも表示しています。

**Fig. 12-12** ソードの当たり判定

**Fig. 12-13** 当たり判定処理の方法

## 軌跡の描画

ソードに関するもう1つのポイントは、ソードの軌跡を描画する方法です。軌跡を描画することによって、ソードの動きが強調され、見た目が非常に派手な攻撃になります。

軌跡を描画するには、ソードが過去に通った座標を記録しておきます。そして、この座標を使って帯状の軌跡を描画します（Fig. 12-14）。記録する座標の数を増やすほど、軌跡は長くなります。

より具体的には、Fig. 12-15のように軌跡を描きます。過去にソードが通った位置について、ソードの両端の座標を記録しておきます。そして、これらの座標を順番に並べて頂点データとし、辺を共有する三角形の連鎖（トライアングルストリップ）を描きます。

Fig. 12-15の場合には、まず頂点「0,1,2」からなる三角形を描画し、次に頂点「1,2,3」からなる三角形を描画します。同様に、頂点「9,10,11」からなる三角形までを順番に描画していきます。

Direct3Dを使うと、こういったトライアングルストリップを簡単に描くことができます。頂点データをまとめて作成しておき、IDirect3DDevice::DrawPrimitiveUP関数にD3DPT_TRIANGLESTRIPを指定して呼び出せば、1回の呼び出しで軌跡を描画することが可能です。

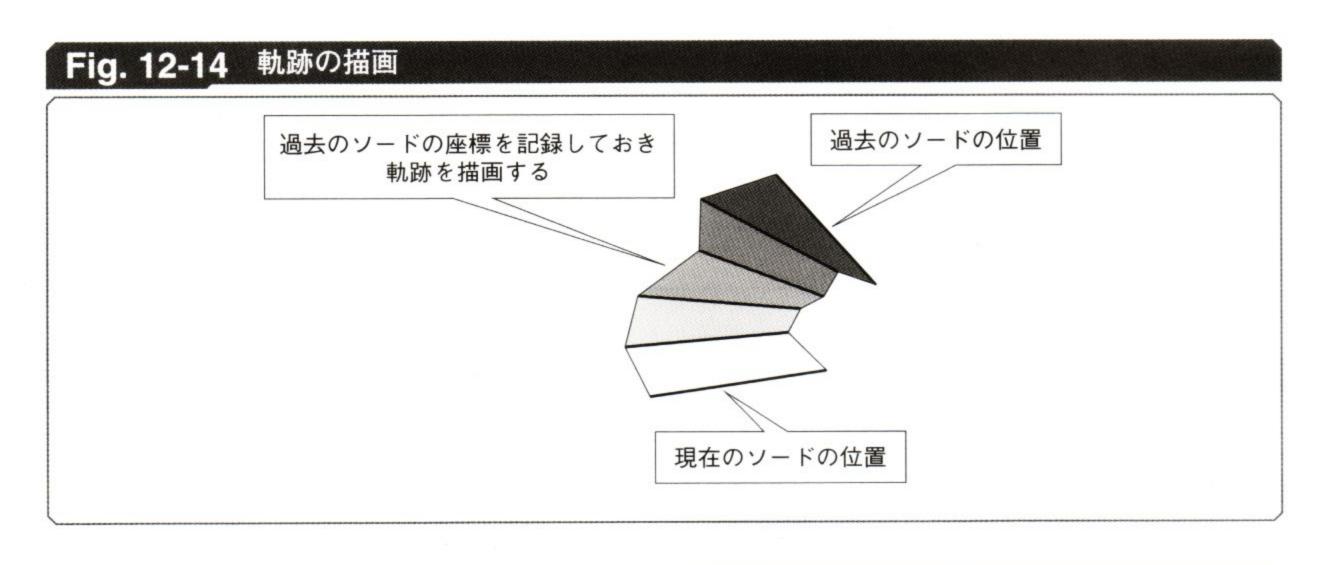
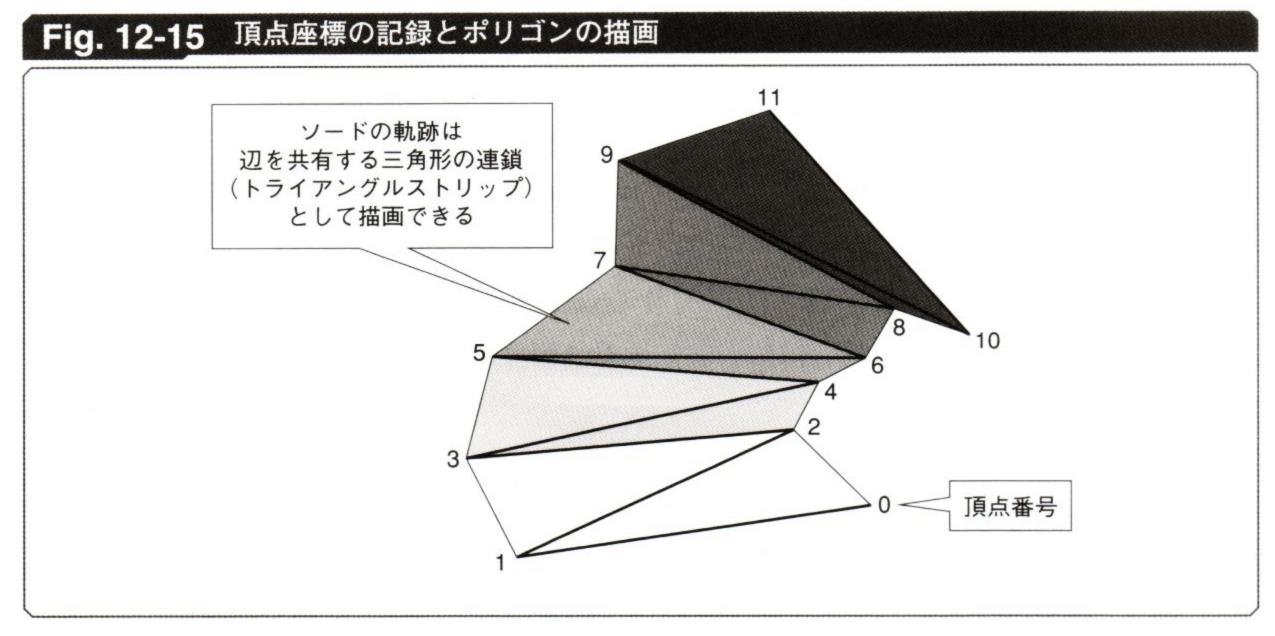
軌跡を描画する際には、アルファブレンディングを使って、ソードがより過去に通った部分ほど薄く描くとよいでしょう。こうすると、軌跡がソードから尾を引くように伸び、末端からしだいに消えていく効果が得られます。

さらに、アルファブレンディングの加算合成を使えば、ソードから光の帯が伸びていく

ような効果を得ることができます。加算合成は、ヘッドライトなどの光る物体を表現するときによく使われる手法です。

軌跡を描画するときには、ポリゴンの裏表に関する設定も必要です。通常の3Dグラフィックでは、物体の裏面を描かないために、裏返ったポリゴンを描画しない設定になっています。しかしソードの軌跡の場合には、裏表にかかわらず軌跡を構成するポリゴンをすべて描画したいので、特別な設定が必要です。

**Fig. 12-14 軌跡の描画**

**Fig. 12-15 頂点座標の記録とポリゴンの描画**

## ソードのプログラム

List 12-1は、ソードに関するプログラムです。ソードの動きは、回転に関する計算と、座標の記録がポイントです。描画に関しては、頂点データの作成処理と、加算合成やポリゴンの裏表に関するデバイスの設定方法に注目してください。

ソードは最初は右回りに1回転し、それから自機の動きに応じて動くようになります。List 12-1では、ソードの状態を表す変数（SwordState）を使い、状態によってソードの動きを変えています。

**List 12-1** ソード（MyShip.h、MyShip.cpp）

```
// 自機の基本クラス
class CMyShip : public CMover {

    // ...（中略）...

    // ソードの状態
    int SwordState;

    // ソードの目標角度、現在角度
    float SwordNewAngle, SwordAngle;

    // ソードと軌跡の頂点座標
    D3DXVECTOR2 SwordPos[SWORD_TRAILS*2];

    // ...（中略）...

};

// 自機の移動
bool CMyShip::Move() {

    // ...（中略）...

    // ボタンを離したらソードを引っ込める
    if (!pi.Button(0)) {
        for (int i=0; i<SWORD_TRAILS*2; i++) {
            SwordPos[i].x=X;
            SwordPos[i].y=Y;
        }
        SwordState=0;
    }

    // ボタンを押したらソードを出す
```

```
if (SwordState==0) {
    if (pi.Button(0)) {
        SwordAngle=0;
        SwordState=1;
    }
}

// ソードが出ているときの処理
else {

    // ソードを回転させる
    if (SwordState<=51) {
        SwordNewAngle=SwordAngle=(SwordState-1)*0.02f;
        SwordState++;
    }

    // ソードを自機に追随させる
    else {

        // 目標角度を決定
        if (pi.Left()) {
            SwordNewAngle=pi.Up() ?
                0.375f : (pi.Down()?0.125f:0.25f);
        } else
        if (pi.Right()) {
            SwordNewAngle=pi.Up() ?
                0.625f : (pi.Down()?0.875f:0.75f);
        } else {
            SwordNewAngle=pi.Up() ?
                0.5f : (pi.Down()?0.0f:SwordNewAngle);
        }

        // 現在角度を更新
        static const float Speed=0.1f;
        float diff=SwordNewAngle-SwordAngle+1;
        diff-=floorf(diff);
        if (diff<0.5f) {
            SwordAngle=SwordAngle+diff*Speed;
        } else {
            SwordAngle=SwordAngle+(diff-1)*Speed+1;
        }
    }

    // ソード両端の座標を登録
    for (int i=SWORD_TRAILS*2-1; i>=2; i--) {
        SwordPos[i]=SwordPos[i-2];
```

```
}
static const float Near=5, Far=30;
float rad=D3DX_PI*2*SwordAngle, s=sin(rad), c=cos(rad);
SwordPos[0].x=X+s*Near;
SwordPos[0].y=Y-c*Near;
SwordPos[1].x=X+s*Far;
SwordPos[1].y=Y-c*Far;

// 弾を斬る
static const float
    HitDistance=1200, HitCos=0.97f, Attack=50;
bool hit=false;
for (CTaskIter i(Game->BulletList); i.HasNext(); ) {
    CBullet* bullet=(CBullet*)i.Next();

    // 距離と角度が一定範囲内ならば接触したとする
    float dx=bullet->X-X, dy=bullet->Y-Y, d=dx*dx+dy*dy;
    if (d<HitDistance) {
        d=sqrt(d)*HitCos;
        if (-s*dx-c*dy>d || s*dx+c*dy>d) {

            // 弾を消去する
            bullet->Crash();
            i.Remove();

            // 効果音を鳴らすフラグをセットする
            hit=true;
        }
    }
}

// 敵を斬る
for (CTaskIter i(Game->EnemyList); i.HasNext(); ) {
    CEnemy* enemy=(CEnemy*)i.Next();

    // 距離と角度が一定範囲内ならば接触したとする
    float dx=enemy->X-X, dy=enemy->Y-Y, d=dx*dx+dy*dy;
    if (d<HitDistance) {
        d=sqrt(d)*HitCos;
        if (-s*dx-c*dy>d || s*dx+c*dy>d) {

            // 敵の耐久力を削る
            enemy->Vit-=Attack;

            // 敵を斬ったエフェクトを出す
            for (int j=0; j<5; j++) {
```

```
                        new CBeamEffect(enemy->X, enemy->Y);
                    }

                    // 効果音を鳴らすフラグをセットする
                    hit=true;
                }
            }
        }

        // 弾や敵を斬ったときに効果音を鳴らす
        if (hit) Game->PlaySE(Game->SEShotBomb);
    }

    // ...（中略）...

}

// 自機の描画
void CMyShip::Draw() {

    // ...（中略）...

    // レンダリング条件の設定
    LPDIRECT3DDEVICE9 d=Game->GetGraphics()->GetDevice();
    d->SetRenderState(D3DRS_ZWRITEENABLE, FALSE);
    d->SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
    d->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
    d->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
    d->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_SELECTARG2);
    d->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG2);

    // 頂点データの作成
    float h=Game->GetGraphics()->GetHeight();
    SWORD_VERTEX v[SWORD_TRAILS*2];
    for (int i=0, k=0; i<SWORD_TRAILS-1; i++) {

        // 過去の位置ほど薄く描くようにアルファ値を設定する
        float c=(float)i/SWORD_TRAILS;
        D3DCOLOR color=D3DXCOLOR(
            1.0f-c*0.5f, 0.5f, 0.3f+c*0.5f, 1.0f-c);

        // 記録した座標から頂点データを作る
        for (int j=0; j<2; j++, k++) {
            v[k].X=(SwordPos[k].x+50)*0.01f*h;
            v[k].Y=(SwordPos[k].y+50)*0.01f*h;
            v[k].Z=0;
```

```
                v[k].RHW=1;
                v[k].Diffuse=color;
            }
        }

        // ソードと軌跡を表すポリゴンの描画
        d->SetFVF(SWORD_D3DFVF_VERTEX);
        d->DrawPrimitiveUP(D3DPT_TRIANGLESTRIP,
            (SWORD_TRAILS-1)*2, v, sizeof(SWORD_VERTEX));

        // レンダリング条件の復帰
        d->SetRenderState(D3DRS_ZWRITEENABLE, TRUE);
        d->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
        d->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
        d->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
        d->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);
        d->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_MODULATE);

        // ...（中略）...

}
```

## 特殊攻撃を追加する②「ワイパー」

もう1つ、特殊攻撃の例を紹介しましょう。ソードと似た斬り攻撃ながら、自機の動きとは関係なく常に一定の動きをする攻撃を作ります。ゲームによってはそれをソードと呼んでいる場合もありますが、本書では前述のソードと区別するために「ワイパー」と呼ぶことにします。

サンプルでは、タイトル画面で「START (WIPER)」を選択すると、ワイパー攻撃のみのモードを遊ぶことができます。ワイパーを出すには、キーボードの「Z」キーまたはジョイスティックのボタン「0」を押しっぱなしにします。ボタンを離すと、ワイパーは引っ込みます。

ボタンを押すと、ワイパーが自機の正面に出現します。サンプル「紫雨」は寿司をモチーフにしたゲームなので、ワイパーは割り箸を模したデザインにしてみました。

ボタンを押しっぱなしにしていると、ワイパーは自機の左右に回転しながら開きます。ワイパーは自機の左右斜め後方まで開き、少し時間がたつと消えます。さらにボタンを押していると、再び自機の正面にワイパーが出現します (Fig. 12-16)。

ワイパーで弾や敵を斬ると、弾を消したり、敵を破壊したりすることができます。敵を斬ったときには、液体（醤油）が飛び散るエフェクトを表示し、効果音も再生します。

市販のゲームにも、このような近接攻撃を採用したゲームがけっこうあります。例えば、「式神の城」シリーズ（アーケード/ドリームキャスト/GC/PC/PS2/Xbox）、「カオスフィールド」（アーケード/GC/PS2）、「ラジルギ」（アーケード/GC/PS2）、「HOMURA」（アーケード/PS2）などには、弾や敵を斬ることのできる近接攻撃が登場します。前述の「レイディアントシルバーガン」におけるソードとの違いは、自機の動きにかかわらず、近接攻撃の動きが一定だということです。

**Fig. 12-16** ワイパー

## ワイパーの仕組み

ワイパーとソードの仕組みは似ていますが、ワイパーの方がだいぶん簡単です。ソードの場合には角度の調整や座標の記録が必要ですが、ワイパーの場合には必要ありません。

ワイパーを実現するには、タイマーとなる変数を用意します。そして、タイマーの値に応じて、最初は自機の正面にワイパーを出現させ、時間とともにワイパーを左右に回転させて開きます。一定時間が経過したらワイパーの回転を止めて、ワイパーを消去します。

ワイパーをなめらかに出したり消したりするには、アルファブレンディングを使うとよいでしょう。また、ワイパーが通った軌跡を描くことによって、ワイパーの動きを強調することができます。ワイパーは規則的に回転するだけなので、ソードのように座標を記録しておかなくても、計算で過去の座標を求めて軌跡を描くことができます。

ワイパーの当たり判定処理には、ソードの場合と同じ手法が使えます。ワイパーを中心とする扇形の領域を当たり判定にすれば、ワイパーで弾や敵を斬るような感覚を表現する

ことができます。

List 12-2は、ワイパーに関するプログラムです。ワイパーの状態を表す変数（WiperState）は、ワイパーの動きを切り替えるためのタイマーとして使用します。

**List 12-2　ワイパー（MyShip.h、MyShip.cpp）**

```
// 自機の基本クラス
class CMyShip : public CMover {

    // ...（中略）...

    // ワイパーの状態
    int WiperState;

    // ワイパーの角度、アルファ値
    float WiperAngle, WiperAlpha;

    // ...（中略）...

};

// 自機の移動
bool CMyShip::Move() {

    // ...（中略）...

    // ワイパーが出ていないとき：
    // ボタンを押したらワイパーを出現させる
    if (WiperState==0) {
        if (pi.Button(0)) {
            WiperState=1;
            WiperAngle=0;
            WiperAlpha=0;
        }
    }

    // ワイパーが出ているとき
    else {
        // 自機の正面に出現する
        if (WiperState<10) {
            WiperAlpha+=0.1f;
        } else

        // 左右に回転して開く
        if (WiperState<50) {
```

```
WiperAngle+=0.01f;

// 弾を斬る
float rad=D3DX_PI*2*WiperAngle,
    c=cos(rad), s=sin(rad);
static const float
    HitDistance=1200, HitCos=0.97f, Attack=50;
bool hit=false;
for (CTaskIter i(Game->BulletList); i.HasNext(); ) {
    CBullet* bullet=(CBullet*)i.Next();

    // 距離と角度が一定範囲内ならば接触したとする
    float dx=bullet->X-X, dy=bullet->Y-Y,
        d=dx*dx+dy*dy;
    if (d<HitDistance) {
        d=sqrt(d)*HitCos;
        if (-s*dx-c*dy>d || s*dx+c*dy>d) {

            // 弾を消す
            bullet->Crash();
            i.Remove();

            // 効果音を鳴らすフラグをセットする
            hit=true;
        }
    }
}

// 敵を斬る
for (CTaskIter i(Game->EnemyList); i.HasNext(); ) {
    CEnemy* enemy=(CEnemy*)i.Next();

    // 距離と角度が一定範囲内ならば接触したとする
    float dx=enemy->X-X, dy=enemy->Y-Y,
        d=dx*dx+dy*dy;
    if (d<HitDistance) {
        d=sqrt(d)*HitCos;
        if (-s*dx-c*dy>d || s*dx+c*dy>d) {

            // 敵の耐久力を減らす
            enemy->Vit-=Attack;

            // エフェクトを表示する
            for (int j=0; j<5; j++) {
                new CBeamEffect(enemy->X, enemy->Y);
            }
```

```
                    // 効果音を鳴らすフラグをセットする
                    hit=true;
                }
            }
        }

        // 弾や敵を斬ったときに効果音を鳴らす
        if (hit) Game->PlaySE(Game->SEShotBomb);
    } else

    // ワイパーが開ききったらなめらかに消去する
    if (WiperState<60) {
        WiperAlpha-=0.1f;
    } else {
        WiperState=-1;
    }

    // 状態の更新
    WiperState++;
    }

    // ...(中略)...

}

// 自機の描画
void CMyShip::Draw() {

    // ...(中略)...

    // ワイパーが出ていたら描画する
    if (WiperState!=0) {

        // 左右のワイパーを描画するためのループ
        for (int i=-1; i<=1; i+=2) {
            CMesh* mesh=Game->MeshChopSticks[(i+1)/2];

            // ワイパーと残像を描画するためのループ
            for (float angle=WiperAngle,
                alpha=WiperAlpha;
                angle>=0 && alpha>0; angle-=0.01f, alpha*=0.9f
            ) {
                // 色とアルファ値の設定
                mesh->SetColor(
                    D3DXCOLOR(1, 1, 1, alpha),
```

```
                D3DXCOLOR(0, 0, 0, 0));

            // 3Dモデルの描画
            float rad=D3DX_PI*2*angle*i;
            mesh->Draw(
                D3DXVECTOR3(X+sin(rad)*16, Z, -Y+cos(rad)*16),
                D3DXVECTOR3(2, 2, 2),
                D3DXVECTOR3(0, angle*i, 0), TO_ZYX);
        }
    }
}

// ...（中略）...

}
```

## >>Chapter 12のまとめ

本章では、サンプルゲームをアレンジしてオリジナルゲームを制作するためのヒントを紹介しました。また、ソードやワイパーといった特殊攻撃の実現方法を解説し、これらの攻撃を使って元とはまったく違ったゲーム性を作り出す例も示しました。

もしゼロからオリジナルのシューティングゲームを作るのは大変だと感じていたら、本書のサンプルを利用して、グラフィック・サウンド・スクリプトといったデータを変更するところから始めてみるのがお勧めです。次に、弾・敵・特殊攻撃といった部分的なプログラムを、変更したり新規に書いたりしてみるとよいでしょう。また、例えば横スクロールゲームにアレンジするといったように、ゲームの形式を大きく変更してみても面白そうです。

# Appendix

#  ゲームのブラッシュアップ

本書のサンプルには、ここまでに解説してきた機能の他に、リプレイ機能や難易度の選択といった、ゲームをより楽しく遊ばせるための工夫が盛り込まれています。また、付録CD-ROMには、ゲームをリリースする際にデータをアーカイブ化するためのツールも収録しています。

各機能については、付録CD-ROMに収録した「ChapterA.pdf」で解説しているので、詳しくはそちらをご参照ください。ここでは、概要について簡単に触れておきます。

## ◆ リプレイ機能

リプレイは、プレイヤーがプレイしたゲームの内容をビデオのように保存しておき、あとから再生する機能です。リプレイ機能があると、上手にプレイしたときの記録を残しておいたり、自分のプレイを見直して攻略に役立てたりすることができます。また、他人に自分のプレイを見てもらったり、他人のプレイを見せてもらったりすることも可能なので、ゲームの楽しみ方が広がります。

サンプルのタイトル画面で「REPLAY」を選択すると、直前のプレイ内容を再生することができます。

## ◆ 難易度の選択

一般にゲームの難易度は、プレイヤーの腕前と釣り合いが取れているか、ほんの少しだけ難しいくらいが面白いものです。簡単すぎても、逆に難しすぎても、ゲームの面白さを損なってしまいます。プレイヤーの技量には個人差があります。また、同じプレイヤーでも、ゲームに慣れれば上手になるでしょう。こういった技量の差を吸収する方法の1つは、ゲームに難易度の選択機能をつけることです。

本書のサンプルでは、「NORMAL」「HARD」「MANIAC」といった3段階の難易度の設定が可能になっています。

タイトル画面で難易度を選べます（Fig. A-1）。

## ◆ データファイルのアーカイブ化

ゲームでは画像やサウンドなど、さまざまなデータファイルを使います。ゲームをリリースする際に、これらのデータファイルをそのまま収録してもよいのですが、データファイルがそのまま入っていると、画像ツールやサウンドツールなどを使って、データを簡単

に開くことができてしまいます。

また、非常に小さなデータファイルが数多くある場合には、ディスク領域を無駄に消費するおそれがあります。ファイルをロードする際にも、多数の小さなファイルを読み込むのは、少数の大きなファイルを読み込むよりも時間がかかりがちです。

このような問題点を解決するには、複数のデータファイルをアーカイブ化して、1つのファイルにまとめる方法があります。さらに暗号化を行えば、データファイルが意図していない方法で利用される可能性も少なくなります。

付録CD-ROMの「Archiver」フォルダには、そのためのツールとプロジェクトファイル一式が収録されています。ツールの使い方などは、「ChapterA.pdf」をご参照ください。

**Fig. A-1** 難易度の選択

# Visual C++ 2005 Express Editionで Win32アプリケーションを作成する方法

Visual C++ 2005 Express Edition（以下、Express Edition）は無償で利用できる開発環境です。有償で販売される上位のエディションに比べると機能に制限はあるものの、最新版のVisual C++ 2005が無償で使えるということで、大変魅力的な製品だといえます。

本書のようなシューティングゲームのプログラミングにも、このExpress Editionを利用することができます。ただし、普通にインストールした状態では、Express EditionでWin32アプリケーションを作ることはできません。.NET Frameworkアプリケーションは作れるのですが、Win32アプリケーションの作成機能は無効になっているのです。

幸い、MSDNに含まれる下記のドキュメントでは、Express EditionでWin32アプリケーションを作成するための設定方法が公開されています。同製品をお使いの方のために、本書でもこの設定方法を紹介します。

**Visual C++ 2005 Express EditionとMicrosoft Platform SDKを一緒に使う**
http://www.microsoft.com/japan/msdn/vstudio/express/visualc/usingpsdk/

## ◆ 開発環境の準備

開発環境をインストールして、Win32アプリケーションの作成に必要な設定を行います。

### ①Visual C++ 2005 Express Editionのインストール

Express Editionをダウンロードして、インストールします。入手方法とインストール方法は、下記のドキュメントに記載されています。

**Visual C++ 2005 Express Edition 日本語版**
http://www.microsoft.com/japan/msdn/vstudio/express/visualc/

### ②Microsoft Platform SDKのインストール

Microsoftのダウンロードセンターを利用して、x86プラットフォーム用の「Windows Server 2003 SP1 Platform SDK」をWeb経由でインストールします。

**ダウンロードセンター**
http://www.microsoft.com/downloads/

### ③パスの設定

Platform SDKのインクルードパスやライブラリパスを登録します。Express Editionを起動して、［ツール］→［オプション］メニューで［オプション］ダイアログを開き、［プロジェクトおよびソリューション］の［VC++ディレクトリ］に、以下のようなパスを追加します。これは「C:¥Program Files¥Microsoft Platform SDK」にPlatform SDKをインストールした場合の例です。

・実行可能ファイル
　C:¥Program Files¥Microsoft Platform SDK¥Bin
・インクルードファイル
　C:¥Program Files¥Microsoft Platform SDK¥include
・ライブラリファイル
　C:¥Program Files¥Microsoft Platform SDK¥lib

### ④ライブラリの追加

Win32アプリケーションに必要なライブラリを追加します。下記のファイルをテキストエディタ（メモ帳など）で開きます。

C:¥Program Files¥Microsoft Visual Studio 8¥VC¥
VCProjectDefaults¥corewin_express.vsprops

そして、以下の部分を変更します。

・変更前
　AdditionalDependencies="kernel32.lib"
・変更後
　AdditionalDependencies="kernel32.lib user32.lib gdi32.lib winspool.lib
comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib"

### ⑤Win32テンプレートの有効化

Win32アプリケーション用のテンプレートを有効化します。下記のファイルをテキストエディタで開きます。

C:¥Program Files¥Microsoft Visual Studio 8¥VC¥
VCWizards¥AppWiz¥Generic¥Application¥html¥1041¥AppSettings.htm

そして、ファイルの441～444行目を以下のようにコメントアウトします。

```
// WIN_APP.disabled = true;
// WIN_APP_LABEL.disabled = true;
// DLL_APP.disabled = true;
// DLL_APP_LABEL.disabled = true;
```

## ◆ Win32アプリケーションの作成例

これで開発環境の設定は終わりです。正しく設定ができたかどうかを確かめるためには、次のように簡単なWin32アプリケーションを作成してみるとよいでしょう。

### ①Visual C++ 2005 Express Editionの起動

設定ファイルを変更したので、開発環境を再起動する必要があります。

### ②プロジェクトの作成

メニューの［ファイル］→［新規作成］→［プロジェクト］を選択し、［新しいプロジェクト］ダイアログを開きます。「プロジェクトの種類」ツリーの［Visual C++］→［Win32］を選択し、「テンプレート」から［Win32 コンソール アプリケーション］を選択します。そして任意のプロジェクト名を指定し、［OK］ボタンをクリックします（Fig. A-2）。

### ③ウィザードの操作

「Win32アプリケーションウィザード」が表示されます。Win32アプリケーションを作成するには、「アプリケーションの種類」で［Windowsアプリケーション］を選択します。雛型となるアプリケーションを生成させない場合には、［空のプロジェクト］をチェックします。そして、最後に［完了］ボタンをクリックします（Fig. A-3）。

### ④アプリケーションの実行

［空のプロジェクト］をチェックしなかった場合には、簡単なアプリケーションが生成されます。［ビルド］→［ソリューションのビルド］と［デバッグ］→［デバッグ開始］を選択すれば、アプリケーションを構築して実行することができます（Fig. A-4）。

Fig. A-2 新しいプロジェクトの生成

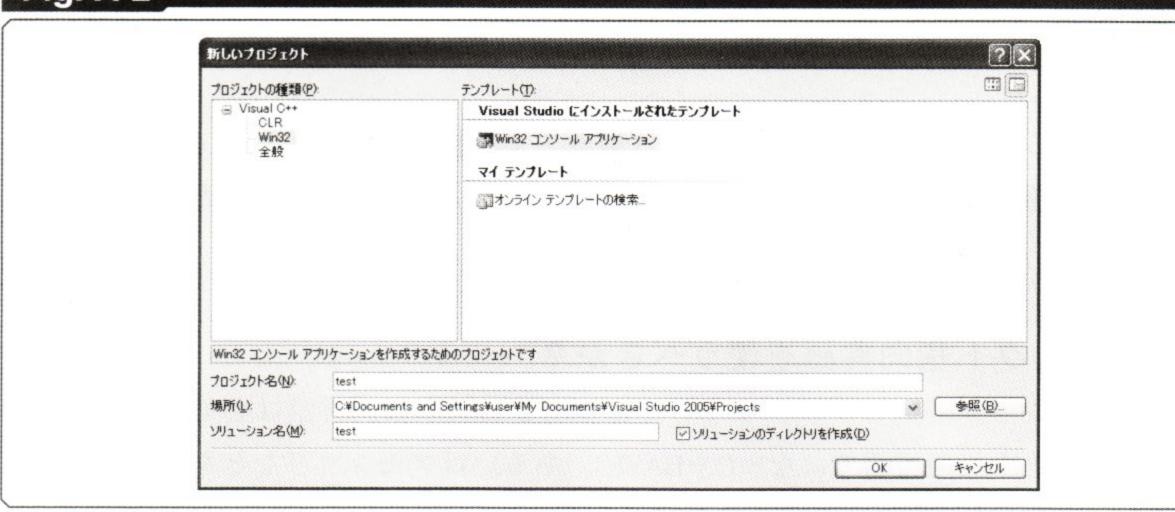



Fig. A-3 Win32アプリケーションウィザード

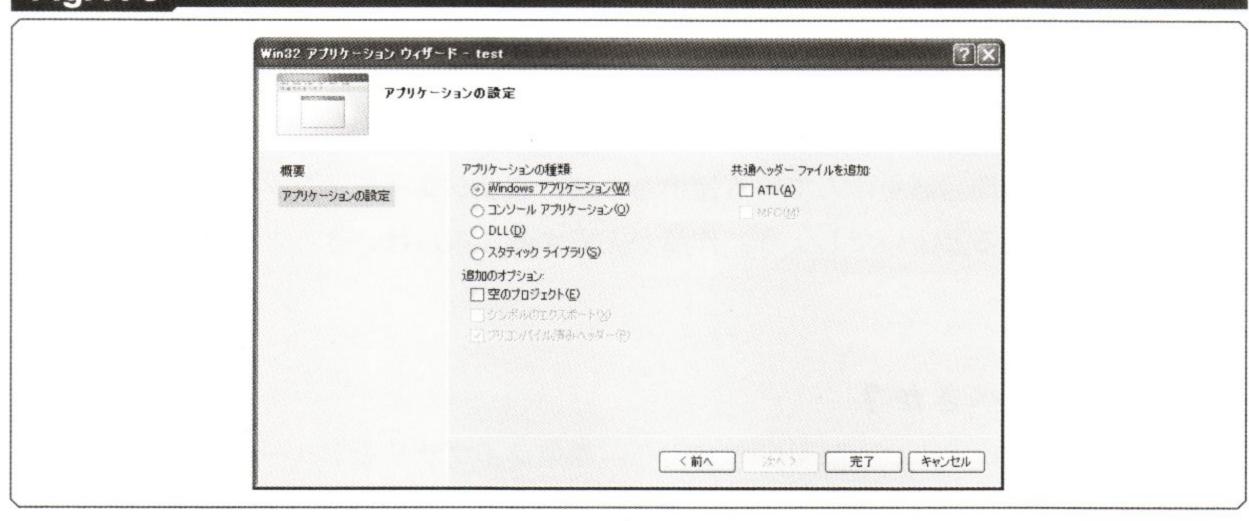
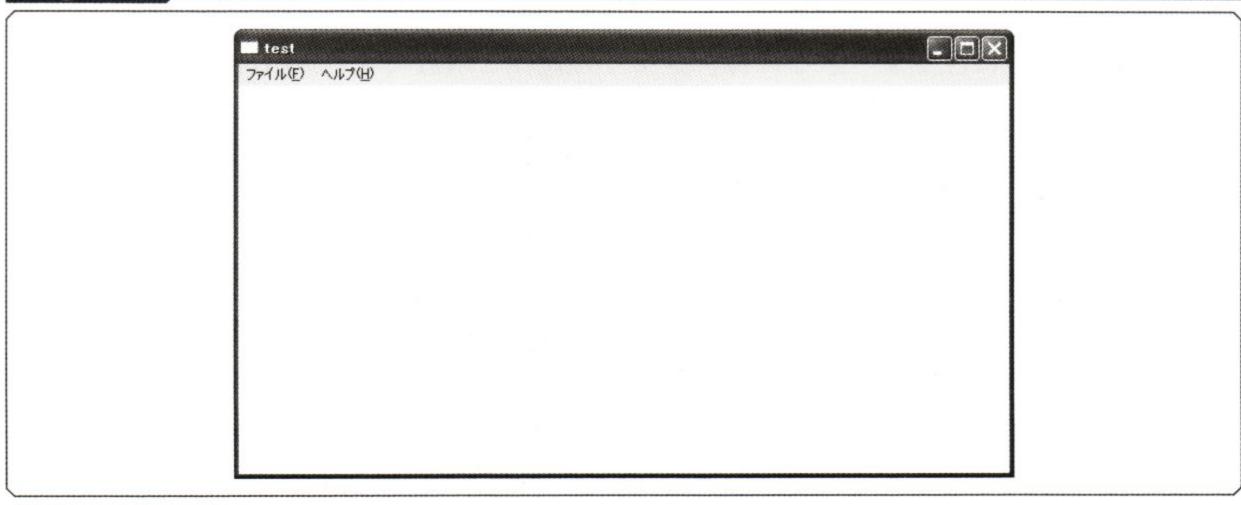



Fig. A-4 アプリケーションの実行画面

#  シューティングゲームプログラミングQ&A

ここでは、シューティングゲームのプログラミングに関するいくつかのポイントを、Q＆A形式で紹介します。本書を読み解くためのヒントとしてお役立ていただければ幸いです。

## ◆オブジェクト指向言語を使うべきか？

シューティングゲームのプログラミングにオブジェクト指向言語は必須ではありません。オブジェクト指向言語ではないCやBASIC、あるいはアセンブリなどを使っても、問題なくシューティングゲームを作ることができます。

ただ、C++などのオブジェクト指向言語を知っているならば、ぜひ使うことをお勧めします。シューティングゲームは、自機・弾・敵・武器といったオブジェクトにまとめやすい要素から構成されています。オブジェクト指向言語を使うと、キャラクターの種類ごとにデータや処理をきれいにまとめたり、名前空間を整理したりすることが簡単にできます。

オブジェクト指向言語を学んでいる途中ならば、ぜひシューティングゲームを練習台に言語の学習を進めるとよいでしょう。作品が完成するころには、きっと言語を深く理解できているはずです。

## ◆STLは使うべきか？

STL（Standard Template Library）つまりC++標準ライブラリをシューティングゲームのプログラミングに使うべきかどうかは、場合によります。C++標準ライブラリのvectorやlistといったクラスはとても便利ですが、処理の性質によってはこれらのクラスを使わずに、独自の方法でデータの管理を行った方がよいこともあります。

STLを使うときに注意すべきなのは、メモリの確保と解放です。vectorやlist、あるいはstringなど多くのクラスは、必要が生じると自動的に追加のメモリを確保します。メモリの確保と解放に関するアルゴリズムはライブラリまかせなので、メモリに関するプログラムの挙動をプログラマが正確に予測することは困難です。

そのため、例えばタスクシステムのように頻繁にメモリの確保と解放を繰り返す処理では、独自の方法で管理を行う方が、プログラムの挙動がはっきりとわかるという点で安心できます。一方で、ちょっとしたメッセージの表示やリソースの格納のように、メモリの確保と解放が激しく行われることがなく、使用量の上限も比較的小さい場合には、STLのようなライブラリを使っても問題ありません。

### ◆ デザインパターンは使うべきか？

デザインパターンをシューティングゲームのプログラミングに使ってもかまわないのですが、必須ではありません。デザインパターンはプログラムの再利用を促進するための技術です。処理を高速化したり、メモリの使用効率をよくしたりするための技術ではありません。それどころか、パターンを適用することによって処理が遅くなったり、メモリを余分に消費したりすることも珍しくないのです。

あなたがシューティングゲームをプログラミングするうえで重要視しているのは、プログラムの再利用を促進することでしょうか？　あるいは速度やメモリの使用効率を向上することでしょうか？　場合に応じてきちんと目的を考えながら、パターン適用の是非を判断する必要があります。

デザインパターンについて学ぶには、下記の参考文献が役立ちます。この本はデザインパターンに関する原本の邦訳版です。同書は各パターンのメリットとデメリットを明示しているので、パターンの使いどころを的確に判断するための助けになるでしょう。

『オブジェクト指向における再利用のためのデザインパターン』
　Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 著
　本位田 真一、吉田 和樹 監訳
　ソフトバンククリエイティブ、ISBN 4-7973-1112-6

### ◆ ガベージコレクションは使うべきか？

ガベージコレクションは、不要になった使用ずみメモリを自動的に再利用するための仕組みです。ガベージコレクションがあると、メモリの確保と解放を気軽に行うことができます。言語処理系がガベージコレクションの機能を含んでいる場合もあります。

しかし、シューティングゲームのプログラムでは、基本的にガベージコレクションは使わない方がよいでしょう。ガベージコレクションは手軽な反面、メモリを再利用する処理に時間がかかったり、メモリの正確な消費量が読みにくいといった問題を抱えています。

タスクシステムのように、メモリの確保と解放を何度も繰り返すような処理では、処理時間やメモリ消費量が不透明なガベージコレクションに頼るのは危険な場合があります。Chapter 3で解説したように、処理時間やメモリ消費量が正確に予測できる方法でメモリを管理した方が、トラブルが生じにくいといえます。

### ◆ リソースの解放はきちんと行うべきか？

3Dモデルやテクスチャといったリソースは、プログラムの終了時にはすべて解放する

のが正しい作法です。しかし本書のサンプルには、簡単のためにこういった解放処理を省略しているものがあります。

リソースの多くは、プログラムの終了時にOSやライブラリが自動的に解放してくれるため、解放処理を省略しても問題は生じません。解放処理を書いた方がきちんとしたプログラムになりますが、開発の途中段階などではとりあえず省略するのも1つの方法です。

### ◆乱数の種を一定にする理由は？

リプレイ機能を実現するには、リプレイデータといっしょに乱数の種を保存しておくか、乱数の種を常に一定にしておく必要があります（リプレイについては付録CD-ROMのChapterA.pdfで解説）。これは弾や敵の挙動を乱数で制御しているため、リプレイの記録時と再生時で乱数の種を一致させないと、正しくリプレイが行えないためです。

乱数の種を一定にしておくと、リプレイ以外でも利点があります。例えば、乱数によってゲームの難しさやスコアの稼ぎやすさが変化するのを避けて、公平感を増すことができます。乱数の種によってゲームの有利不利が大きく変化してしまうと、都合のよい乱数の種を引けるかどうかが勝負、という偶然性の高いゲームになってしまうからです。

### ◆デストラクタを仮想関数にする理由は？

デストラクタを仮想関数にするのは、派生クラスのインスタンスを破棄したときに、派生クラスのデストラクタが呼び出されるようにするためです。例えば、Chapter 3のタスククラス（CTask）では、デストラクタを仮想関数にしています（P. 90）。

移動物体クラス（CMover）はタスククラスの派生クラスです。移動物体クラスのインスタンスを破棄したときには、まず移動物体クラスのデストラクタを呼び出し、次にタスククラスのデストラクタを呼び出す必要があります。こういった場合には、デストラクタを仮想関数にします。

デストラクタを仮想関数にしないと、移動物体のインスタンスを破棄したときにも、タスククラスのデストラクタだけが呼び出されます。正確には、プログラムに記述された見かけ上の型に対応するデストラクタが呼び出されます。

本書のプログラムでは、弾クラス（CBullet）や敵クラス（CEnemy）といったさまざまなクラスのインスタンスを、タスククラスや移動物体クラスに変換して管理しています。このときデストラクタが仮想関数でないと、プログラム上に書かれたタスククラスや移動物体クラスのデストラクタだけが呼び出され、弾クラスや敵クラスのデストラクタは呼び出されません。デストラクタを仮想関数にするのは、これら派生クラスのデストラクタを正しく呼び出すためです。

#  引用ゲームの紹介

本書内では、いくつかのゲームを解説に引用しています。ここで、引用したゲームについて紹介しておきましょう。PCや家庭用ゲーム機に移植されている作品も多いので、気になるゲームはぜひ一度実際に遊んでみることをお勧めします。

なお、一部のゲームには続編が出ていたり、移植の際にタイトルが変わっていたりする場合がありますので、詳しくはメーカーWebサイトやショップなどでお確かめください。

※凡例

**■ゲーム名**

・メーカー

・年度

・プラットフォーム（AC：アーケード、DC：ドリームキャスト、GC：ゲームキューブ、PC：PC/AT互換機（Windowsパソコン）、PS：プレイステーション、PS2：プレイステーション2、SS：セガサターン、XB：Xbox）

・コメント

**■斑鳩（いかるが）**

・トレジャー（http://www.treasure-inc.co.jp/）

・2001

・AC/DC/GC

・自機の属性を切り替えて弾を吸収する、という独特のルールを採用したゲーム。白と黒を基調とした美しいグラフィック、迫力あるオーケストラサウンド、戦略性とパターン性の強いゲーム内容が特徴。

**■エスプガルーダ**

・ケイブ（http://www.cave.co.jp/）

・2003/2005

・AC/PS2

・ボタン操作によって、処理落ちが発生したかのように弾や敵の動きをスローにすることができる。処理落ちをゲームシステムに取り込んだという点が新鮮なゲーム。続編として「エスプガルーダⅡ」がある。

■カオスフィールド

・エイブル/マイルストーン（http://www.mile-stone.co.jp/）

・2004

・AC/DC/GC/PS2

・ソードで弾を消せたり、フィールドチェンジで難易度を任意のタイミングで切り替えられたりと、特殊な要素がいろいろと盛り込まれたゲーム。ザコとの戦闘がなく、ボス級の大きな敵との戦闘のみでステージが構成されていることも独特。

■ギガウィング2

・カプコン/匠（たくみ）（http://www.takumi-net.co.jp/）

・2000

・AC/DC

・弾を跳ね返す「リフレクトフォース」を駆使して戦うゲーム。バリアを張りながらぶあつい弾幕に突っ込んでいく独特のプレイスタイルを持つ。本作には「ボルカノン」と呼ばれるアイテム大量発生現象があり、前作の「ギガウィング」よりも派手になっている。特に4人同時プレイにも対応しているDC版はお勧め。

■グラディウスV

・コナミ/トレジャー（http://www.konami.jp/）

・2004

・PS2

・名作「グラディウス」の続編として久々に登場したゲーム。オプションをボタンでコントロールする要素が入り、プレイスタイルの幅が広がった。グラフィックにもいろいろな工夫が見られ、美しいレーザーの光跡や、液体の表現などは一見の価値がある。また、「レイディアントシルバーガン」や「斑鳩」のトレジャーが開発を担当していることにも要注目。

■サイヴァリア

・サクセス（http://www.success-corp.co.jp/）

・2000/2003

・AC/DC/PS2/XB

・敵弾にかすって自機をレベルアップさせる「BUZZシステム」が特徴のゲーム。自機の当たり判定が極端に小さく、またレベルアップの瞬間は無敵になるので、針穴のような弾

幕の隙間を通り抜けることができる。アレンジ版に「サイヴァリア リビジョン」、続編に「サイヴァリア2」がある。

### ■式神の城（しきがみのしろ）

・アルファシステム（http://www.alfasystem.net/）
・2001/2003/2006
・AC/DC/GC/PC/PS2/XB
・テンポのよいゲーム展開や個性的なキャラクター、豊富な演出などが好評を博したゲーム。続編として「式神の城Ⅱ」と「式神の城Ⅲ」がある。高密度の弾幕やかすりによるパワーアップなど、最近のシューティングらしい要素をひとそろい楽しむことができる。

### ■首領蜂（どんぱち）

・アトラス/ケイブ（http://www.cave.co.jp/）
・1995
・AC/PS/SS
・「怒首領蜂」「怒首領蜂 大往生」などの「蜂」シリーズの原点となるゲーム。後継作品に比べると地味だが、「ボタン連打でショット、押しっぱなしでレーザー」というルールはこのゲームから確立された。

### ■怒首領蜂（どどんぱち）

・アトラス/ケイブ（http://www.cave.co.jp/）
・1997
・AC/PS/SS
・「首領蜂」の続編。ショットとレーザーの撃ち分けはそのままに、ゲーム全体をグレードアップさせた作品。弾幕が厚く難易度も高めだが、ゲームのテンポがよく、当たり判定などの調整も絶妙で、不思議なほど気持ちよく遊ぶことができる。難易度を大幅にアップさせた続編「怒首領蜂 大往生（どどんぱち だいおうじょう）」はPS2に移植されている。

### ■バトルガレッガ

・ライジング/エイティング（http://www.8ing.net/）
・1996
・AC/SS
・難易度調整に特徴があるゲーム。ゲームの進行とともに自動的に上がっていく難易度を下

げるために、「残機を稼いだうえで自爆する」という独特のプレイスタイルが要求される。

### ■HOMURA（ほむら）

・タイトー/SKonec Entertainment（http://www.taito.co.jp/）

・2005

・AC/PS2

・日本の戦国時代を舞台にした和風シューティングゲーム。開発を担当したSKonec Entertainmentは韓国の会社。刀で弾を跳ね返して敵に当てたり、刀で敵を直接斬ったりといった、さまざまな抜刀攻撃が用意されている。

### ■虫姫さま（むしひめさま）

・ケイブ（http://www.cave.co.jp/）

・2004

・AC/PS2

・「風の谷のナウシカ」を想起させる世界観を持つゲーム。敵は巨大な虫型の生物が中心、ケイブらしくぶあつい弾幕が楽しめる。難易度選択があり、特に高難易度のウルトラモードでは、信じられないほど高密度の弾幕がプレイヤーを出迎えてくれる。

### ■RAIDEN FIGHTERS（らいでんふぁいたーず）

・セイブ開発

・1996

・AC

・「ライデン」の名を冠しているものの、同社の「雷電」シリーズとはかなり内容が異なるゲーム。敵弾に自機を近づけると火花が出て、火花が出ている間はスコアが加算されるという「かすりボーナス」を取り入れている。

### ■ラジルギ

・エイブル/マイルストーン（http://www.mile-stone.co.jp/）

・2005

・AC/DC/GC/PS2

・携帯電話を題材にしたシューティングゲーム。ショットのほか、近接攻撃のソードや、敵や弾からエネルギー（電波）を吸収できるアブソネットなどの特殊攻撃がある。トゥーンレンダリングで描かれたグラフィックも特徴。

■レイディアントシルバーガン

・トレジャー（http://www.treasure-inc.co.jp/）

・1998

・AC/SS

・アクションゲームを得意とするトレジャーによる、初の業務用シューティングゲームにして、多くの熱狂的なファンを獲得した作品。7種類の武器を使い分けたり、ソードで弾を消したり、同じ色の敵を連続して倒すことによって武器の経験値を稼いでパワーアップさせたりと、多彩な要素が盛り込まれている。

# 付録CD-ROMについて

本書の付録CD-ROMには、本文内で紹介したサンプルシューティングゲーム「紫雨」のプロジェクトファイルならびに実行ファイル、ゲームライブラリなどが収録されています。

サンプルのプロジェクトは、本文の進行に合わせて処理を追加する形で作成されています。詳しくは、本文の該当箇所をお読みください。

サンプルのフォルダ構成をTable A-1に示します。

Table A-1　サンプルのフォルダ構成

| フォルダ | 内容 |
|---|---|
| LibGame | ゲームライブラリ（Chapter 2） |
| LibUtil | 文字列、乱数などのライブラリ |
| Archiver | アーカイバ※ |
| TaskSystem | タスクシステム（Chapter 3） |
| ShtGame_MyShip1 | 自機の表示と操作（Chapter 4） |
| ShtGame_MyShip2 | ショットとビーム（Chapter 4） |
| ShtGame_Bullet1 | 弾の発射（Chapter 5） |
| ShtGame_Bullet2 | いろいろな弾（Chapter 5） |
| ShtGame_Enemy | 敵の表示と動作（Chapter 6） |
| ShtGame_Scene | ゲームの外枠部分（Chapter 7） |
| ShtGame_Stage | ステージの作成（Chapter 8） |
| ShtGame_Boss | ボスの表示と動作（Chapter 9） |
| ShtGame_Item | アイテムとパワーアップ（Chapter 10） |
| ShtGame_Special | 特殊攻撃の作成（Chapter 11） |
| ShtGame_Arrange | アレンジとブラッシュアップ※（Chapter 12） |

※ブラッシュアップとアーカイバについては付録CD-ROMの「ChapterA.pdf」を参照

## ◆サンプルの実行

サンプルの実行には、DirectX 9.0c以上が動作する環境が必要です。また、プロジェクトのビルドには、DirectX 9.0 SDK（August 2006）が必要となります。DirectXを未入手の方は、マイクロソフトのWebサイト（http://www.microsoft.com/japan/msdn/）などからダウンロードしてください。詳しくは、本文の関連箇所をご参照ください（P. 17）。

また、DirectX 9.0に対応したビデオカードが必要となります。それ以外のビデオカードではサンプルが正しく動作しない場合があります。その点、あらかじめご了承ください。

サンプルの操作方法については、本文の関連箇所をご参照ください（P. 16）。

各サンプルは、付録CD-ROMからハーディスクにフォルダ構成ごとコピーしてお使いください。なお、コピーの際には、「読み取り専用」属性は解除してください。

## ◆ コンピュータ・ウィルスについて

付録CD-ROMに収録したファイル群は、製作関係者による厳密なウィルスチェックを行っております。しかし、それをもってウィルスの不存在を保証するものではありません。各自ウィルスチェックを行い、安全性を確かめることをお薦めします。

すでにウィルスに冒されたコンピュータに付録CD-ROMに収録されたファイルをインストールした場合、付録CD-ROMのファイルもウィルスに冒されてしまう可能性があります。これについては、製作関係者の責任の及ぶところではありません。

## ◆ 著作権について

本書の内容、付録CD-ROM内のファイルは、すべて著作権法上の保護を受けています。本書の内容ならびにファイルの全部または一部を無断で複写・複製・転載することは法律によって禁じられています。

本書内ならびにファイル内に収録されているプログラムソースは、本書籍著作者に著作権がありますが、個人的利用ならびにプログラミングの学習目的としての利用にかぎって、自由に改変して使用することが許可されています。

本書ならびにファイル内に収録されているプログラムを利用したソフトウェアを配布・販売する場合には、あらかじめ著者Webサイトからお問い合わせください。なお、以下の条件をすべて満たしている場合には、お問い合わせいただかなくても、本書のプログラムを利用したソフトウェアを配布することができます。

- 非営利目的で無料で配布するソフトウェアであること（企業・団体・商品などのPRのための無料配布の場合は事前にお問い合わせください）
- ゲーム内およびゲームに添付するドキュメント内において、プログラムの著作権者（松浦健一郎・司ゆき/ひぐぺん工房）を表示すること
- ソースコードを配布しないこと

# INDEX

■D

■E



■F



■G



■I



■M



■N



■O



■P



■Q



■R

■か行



■さ行

■た行

■な行



■は行

# おわりに

C MAGAZINE誌で連載させていただいた「ゲーム・ノ・シクミ～シューティングゲーム編～」の書籍化ということで始まった本書の執筆ですが、サンプルゲームを改良したり、解説を大幅に読みやすくしたり、新しいトピックを追加したりしているうちに、ほとんど書き下ろしといっても差し支えないほどの充実ぶりになりました。お楽しみいただければ幸いです。

なお、古今東西のさまざまなシューティングゲームにおけるギミックを解説した書籍『シューティングゲームアルゴリズムマニアックス』も、ぜひ本書とあわせてご利用ください。3Dグラフィックにも興味をお持ちでしたら、数々の3Dエフェクトを作成する方法が満載の書籍『ゲームエフェクトマニアックス』も、カッコいいシューティングゲームを作るためにきっと役立てていただけるかと存じます。

末筆になりましたが、BGMの使用を快諾していただいたKohzas氏に篤く御礼申し上げます。今後も素晴らしい曲をお作りになられますよう、心から応援しております。

本書がシューティングゲーム制作を始めるきっかけになったり、シューティングゲームについて熱く語ったり、今まで以上にシューティングゲームが好きになったりするための手助けになることを願っています。

松浦 健一郎/司 ゆき

■本書のサポートページ

http://isbn.sbcr.jp/37214/

■著者プロフィール

松浦 健一郎(まつうら けんいちろう)

東京大学工学系研究科電子工学専攻修士課程を修了後、研究所勤務を経て、現在は趣味と実益を兼ねつつフリーのプログラマー&ライターとして活動中。著書に『はじめてのJBuilder4』『Delphi DB&Webプログラミング』『はじめてのJBuilder6』『デスクトップマスコットを作ろう!』『ゲームエフェクト マニアックス』『Javaのココロ』(以上、司ゆきとの共著)、『シューティングゲーム アルゴリズム マニアックス』がある(いずれもソフトバンククリエイティブ刊)。関心と仕事の範囲はプログラミングを中心にコンピュータ全般に及ぶが、最も興味がある分野はプログラミング言語作りとゲーム作り。

司 ゆき(つかさ ゆき)

東京大学理学系研究科情報科学専攻修士課程修了。学部生時代からライターおよびプログラマーの仕事を始める。現在の夢はなめこ狩りにいくこと。

著者Webサイト「ひぐぺん工房」
http://cgi32.plala.or.jp/higpen/gate.shtml

# シューティングゲーム プログラミング

2006年9月30日　初版第一刷発行

著　者・・・・・・・・・・松浦健一郎/司 ゆき

発行者・・・・・・・・・新田光敏
発行所・・・・・・・・・ソフトバンククリエイティブ株式会社
〒107-0052　東京都港区赤坂4-13-13
TEL 03-5549-1201(販売)
http://www.sbcr.jp

印刷・・・・・・・・・・岩岡印刷工業株式会社

装丁・・・・・・・・・・Pocket Beat Graphics
本文デザイン/組版 ・・・クニメディア株式会社



Printed In Japan　ISBN4-7973-3721-4

SoftBank Creative
ソフトバンク クリエイティブ 株式会社

ISBN4-7973-3721-4
C0055 ¥2800E

9784797337211

定価 本体2,800円 +税

1920055028004

シューティングゲームプログラミング
Shooting Game Programming

松浦健一郎／司ゆき 著

SoftBank Creative

シューティングゲーム
Shooting Game Programming
プログラミング
COMPACT disc
収録
サンプルシューティングゲーム「紫雨」
SoftBank Creative
© 2006 SOFTBANK Creative Inc. All rights Reserved